AD-A196 205

RADC-TR-88-42
Final Technical Report
March 1988

DTIC FILE COPY

VERY LARGE PARALLEL DATA FLOW

Honeywell Corporate Systems Development Division

R. Ramnarayan, C. Baker, H. Lu, K. Mikkilineni, J. Richardson,
A. Sheth and S. Yalamanchili

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

DTIC
ELECTE
JUL 0 7 1988
E

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

88 7 05 109

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.
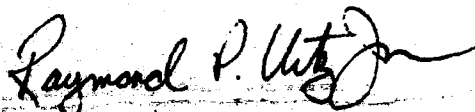
RADC-TR-88-42 has been reviewed and is approved for publication.
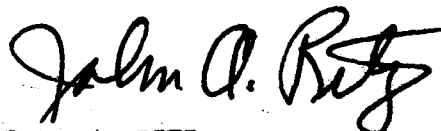
APPROVED:

RAYMOND A. LIUZZI
Project Engineer

APPROVED:

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER:

JOHN A. RITZ
Directorate of Plans & Programs

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A			
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.			
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A						
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-42			
6a. NAME OF PERFORMING ORGANIZATION Honeywell Corporate Systems Development Division		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)			
6c. ADDRESS (City, State, and ZIP Code) 1000 Boone Avenue North Golden Valley MN 55427			7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center		8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0215			
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			10. SOURCE OF FUNDING NUMBERS			
			PROGRAM ELEMENT NO. 62702F	PROJECT NO. 5581	TASK NO 21	WORK UNIT ACCESSION NO. 73

11. TITLE (Include Security Classification)
VERY LARGE PARALLEL DATA FLOW

12. PERSONAL AUTHOR(S)
R. Ramnarayan, C. Baker, H. Lu, K. Mikkilineni, J. Richardson, A. Sheth, S. Yalamanchili

13a. TYPE OF REPORT Final	13b. TIME COVERED FROM Dec 85 TO Dec 87	14. DATE OF REPORT (Year, Month, Day) March 1988	15. PAGE COUNT 276

16. SUPPLEMENTARY NOTATION
N/A

17.	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer Architecture
09	02		Data Base (Computer architecture
			Parallel

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
This report describes a three-phase effort to investigate the use of parallel processing in very large data/knowledge database management systems (D/KBMS) as a means of attaining required performance levels. Phase One involved investigation of parallel processing techniques for inference processing, parallel processing techniques for very large database management and fault tolerant techniques for very large databases. During phase two, the results of six investigation studies were combined to develop a methodology for specifying high performance, highly available D/KBMS architectures for very large data/knowledge base (D/KB) environments. During phase three, a data/knowledge base management testbed was designed and implemented to serve as both a demonstration and performance measurement and evaluation platform. The testbed was used to perform several experiments designed to quantitatively measure D/KBMS performance and understand D/KBMS performance sensitivity and behavior with respect to various system paramounts. Several key results and conclusions have been identified from this work and directions for future work have been identified.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☑ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Raymond A. Liuzzi	22b. TELEPHONE (Include Area Code) (315) 330-2643	22c. OFFICE SYMBOL RADC (COTD)

DD Form 1473, JUN 86   Previous editions are obsolete.   SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This document constitutes the final report for the Very Large Parallel Data Flow (VLPDF) program. This introduction contains the VLPDF program background, program overview, our overall approach, and an outline of the rest of this report, including the key results contained in each chapter.

1.1. Background

Battle management involves managing large volumes of real time data, interpreting this data, correlating data from multiple sources into a multidimensional view of the world, predicting enemy actions, monitoring incoming information for enemy threats and inconsistencies with predictions, and planning effective countermeasures. The large volumes of incoming data and the short response times required will force computers to take over many of the analysis and decision-making functions currently performed by humans. This implies the use of knowledge based techniques to implement these sophisticated functions. Battle management systems will therefore be required to manage large data/knowledge bases.

What does managing a large data/knowledge base mean? What are the functions of a Data/Knowledge Base Management System (D/KBMS)? These questions are being investigated by several research groups. However, Brodie's definition seems to capture the essence of a knowledge base management system. He defines a knowledge base management system as "a system providing highly efficient management of large, shared knowledge bases for knowledge-directed applications" [Brod86]. While there is ongoing debate about the functionality of a D/KBMS, this and other definitions imply that, at a very minimum, a D/KBMS must provide a set of facilities analogous to the data definition, data manipulation, data access, and data integrity facilities provided by a database management system (DBMS).

A D/KBMS is a combination of two different search engines — an inferential search engine and a query evaluation search engine. The key technical challenge in designing a D/KBMS is performance, since an inappropriate combination of these two search engines can lead to very poor response times. The performance issue is particularly significant in a battle management environment, due to the large size of the data/knowledge base and the short response times required. A D/KBMS must deliver very high performance, to be effective in such an environment.

The objective of the VLPDF program was to investigate the use of parallel processing in very large data/knowledge base management as a means of attaining the required performance levels. The program is motivated by the following observations: 1) special

purpose parallel architectures have been shown to provide high performance for large database applications, and 2) database management is the minimum functionality a D/KBMS must provide. Parallel processing techniques have been used in database management to provide substantially higher performance than mainframe based systems for large database applications. For example, Thinking Machine Corporation's 64,000 processor Connection Machine performs five times faster than a Cray mainframe computer in searching a text database of 10 billion characters. Teradata Corporation's DBC/1012, a parallel relational database machine, is another high performance engine for large database applications. The DBC/1012 can be scaled up to 1024 processors, each with a Gigabyte of disk, for a total database size of a Terabyte. Both machines embody special purpose parallel architectures: the Connection Machine, a special purpose architecture optimized for operations on large, complex data structures, and the DBC/1012, a special purpose architecture optimized for relational operations. The emphasis of the VLPDF program was on investigating the use of such special purpose parallel architectures for very large *data/knowledge* base management.

1.2. Program Overview

The VLPDF program spanned 24 months and was divided into three phases.

1.2.1. Phase I

This phase involved investigation of:

- parallel processing techniques for inference processing.
- parallel processing techniques for very large database management, and
- fault tolerance techniques for very large databases.

Both inference processing and database management were included because a D/KBMS is a combination of an inferential search engine and a query evaluation search engine. The investigation included algorithms and architectures for relational database machines, and different forms of parallelism present in inference processing — AND, OR, stream, search, etc.

Fault tolerance was included because a D/KBMS must be highly available, to be effective in a battle management environment. The fault tolerance investigation was to emphasize high data availability techniques for *parallel* data/knowledge base management systems, where the larger number of components involved may render the system more susceptible to failure.

1.2.2. Phase II

This phase also spanned 9 months and involved development of:

- a methodology for specifying various architecture approaches for large data/knowledge base management systems, and
- a set of guidelines for choosing among them.

1.2.3. Phase III

This phase spanned 6 months and involved:

- development of a test plan for evaluating candidate D/KBMS architecture approaches, and

- demonstrating capability to search and update a very large data/knowledge base.

1.3. Overall Approach

One issue that confronted us early in Phase I was the choice of a broad approach to integrating Artificial Intelligence (AI) and database technologies to realize a D/KBMS. There are basically four such approaches:

- *Loose coupling.* Couple an existing AI system (Prolog, Lisp, existing expert system shells, etc.) with an existing DBMS.

- *DBMS extension.* Extend the data model of a DBMS with knowledge representation and inference capabilities.

- *AI system extention.* Include database functionality in an AI system.

- *Tight coupling.* Combine the AI and DBMS concepts of knowledge representation and data modeling, i.e., integrate at the so-called knowledge level.

Keeping in mind the 9 month duration of Phase I, we felt that it was better to choose one broad approach, after carefully considering all four, rather than investigating all four approaches to the same level of detail. Choosing an approach, we felt, would enhance the chances of getting concrete results for the program. The issue confronting was: which of these four broad approaches is best suited to exploiting the capabilities of a parallel search engine?

In the loose coupling approach, the DBMS is used just as a query evaluation search engine, with all inferential search being done by the AI system. The loose coupling approach, therefore, does not exploit the capabilities provided by parallel DBMS architectures for inference processing, and so, will most likely result in a D/KBMS that performs poorly.

The tight coupling approach is the most promising one. However, significant technical challenges must be overcome before a D/KBMS with knowledge level integration becomes feasible. Several research efforts are ongoing that address these challenges. We believe that it is too early to tell how parallel search engines can be exploited in the tight coupling approach.

Out of the remaining two approaches, we felt that the DBMS extension approach was the more promising one for exploiting parallel search engines. Parallel relational database machines, such as the DBC/1012, provide very high performance for large, shared database applications. Admittedly, the Connection Machine, a so-called AI machine, has also been shown to provide high performance in searching large databases. However, its effectiveness for large, *shared* database applications has not yet been established. This may change in the future. But for our work in the VLPDF program, we we shunned the AI extension approach, and instead, chose the DBMS extension

approach.

In the DBMS extension approach, a D/KBMS is viewed as a functional extension of a DBMS. A database contains only data in *extensional* format, or *facts*. A data/knowledge base, in addition, contains knowledge in *intensional* format. The intensional part of the knowlege base is also called the *rule base*. The query language of a DBMS can manipulate the extensional data stored in the DBMS. The query language for a D/KBMS, in addition, has a deductive inference mechanism, which interprets the rules, combining them with the extensional data to infer new facts not explicitly stored. A DBMS uses query compilation and optimization, along with special purpose parallel architectures to attain high performance for large database applications. Likewise, our overall approach to obtaining high performance in large data/knowledge based applications was:

> **Compile queries to the intensional knowledge base, using appropriate optimizations, into a program that executes against the extensional data. For queries to the extensional knowledge base, use DBMS query compilation and optimization techniques. In both cases, use a parallel database machine to execute the compiled program.**

It was in this manner we exploited special purpose parallel search engines for doing deductive inference, or *knowledge directed retrieval*. The broad approach of extending the functionality a DBMS to realize a D/KBMS proved to be very well suited to leveraging the vast amount of research that has gone into parallel database machine architectures.

We adopted the logic programming paradigm for providing the functional extension to a DBMS. Several research projects have adopted this paradigm, including the Advanced Database System project at MCC and the NAIL! project at Stanford. Logic offers several advantages:

- it provides a uniform formalism for data, rules, views, and integrity constraints;

- it is the basis for relational database theory;

- it is amenable to parallel processing;

- it is an adequate basis for implementing other knowledge representations; and

- it has a sound theoretical foundation, which permits the abstract expression of ideas, independent of their implementation.

1.4. Report Summary

This section gives a brief description of the chapters in the rest of this report, including the key results contained in each chapter.

Chapters 2 through 7 describe the results of the various investigation studies we performed under the VLPDF contract. We performed a total of six studies: (1) alternative D/KBMS application interface languages, (2) parallel architectures for such languages, (3) data/knowledge query processing, (4) transitive closure algorithms, (5) parallel database management system architectures, and (6) fault tolerance in very large

database systems.

Chapter 2 describes our investigation of alternatives for a D/KBMS application interface language. A D/KBMS will be required to support intelligent applications such as planning, monitoring, interpretation, diagnosis, and prediction. These applications will typically be expressed in an expert system shell like language in which the D/KBMS query language is embedded. This is exactly analogous to data processing applications expressed in Cobol with embedded SQL. The motivation for studying D/KBMS application interface language alternatives is that the overall performance depends not only on the D/KBMS performance but also on the execution efficiency of the application interface language on a hardware architecture.

We identified three principal requirements for the D/KBMS application interface language: it must be amenable to large scale parallelism, it must be a suitable base for implementing an expert system shell, and it must support efficient non-procedural database access. We considered three language classes: imperative, functional, and logic. We found that no existing language is uniformly superior to all others with respect to our requirements. However, we believe that PARLOG, a parallel logic-based language, represents the best choice among existing languages.

Chapter 3 presents the results of our investigation of parallel architectures for executing PARLOG. The form of parallelism present in PARLOG is called stream-*AND* parallelism, where several processes work concurrently on constructing the solution to a logic based query. We designed a parallel abstract machine for executing PARLOG, which lays bare all the functions needed to execute PARLOG programs as well as the data objects created and manipulated by these functions. We also developed a simulator for the abstract machine, which serves as a tool for collecting data on the execution behavior of PARLOG and analyzing this data.

The principal conclusion from this work is that shared memory greatly facilitates implementing PARLOG's stream-*AND* parallelism and that the key to high performance stream-*AND* parallelism is an efficient shared memory abstraction on a loosely coupled architecture. Our abstract machine described achieves this via a number of optimizations. These optimizations address critical problems in the design of efficient parallel architectures. They address the principal sources of overhead, viz., communication and memory latencies, and synchronization overheads.

We also investigated the feasibility of executing PARLOG programs on the Connection Machine architecture. The conclusion from this work is that a coarse grained, loosely coupled architecture is better suited than the Connection Machine, since the form of parallelism best supported by the Connection Machine is directly opposite to that found in PARLOG.

Chapter 4 presents the background concepts pertaining to data/knowledge base query processing. The two main concepts covered are least fixed point evaluation and data/knowledge base query optimization.

One of the difficult problems in the design of a D/KBMS is how to evaluate recursive queries efficiently. Among the large family of recursive queries, the transitive

closure query forms a very important subset. They are important because (1) a large number of recursive queries can be expressed using transitive closures, (2) most application problems involving recursive queries which we can see now are actually transitive closure queries, and (3) efficient processing of transitive closure queries will provide a sound base for solving more complicated recursive queries. We evaluated several algorithms for computing the transitive closure of a database relation. The results of this evaluation are presented in chapter 5. Based on this investigation, we concluded that it is possible to further optimize transitive closure processing. This led us to develop new strategies for this problem, which we then present.

Chapter 6 presents the results of our investigation of parallel architectures for database management. Basically, our work in this area has been in the area of parallel algorithms for the join operation. The join operation is an important operation for relational database systems, and will become even more important as logic-based inference capabilities are added to these systems. We describe a number of multiprocessor join algorithms. The algorithms use sort-merge and hashing techniques, and are highly parallel and pipelined. The algorithms are designed to execute on a multiprocessor architecture that is parameterized in the degree of memory sharing, so that tightly coupled, loosely coupled, and intermediate architectures can be modeled. Other architectural parameters include the number of processors, number of disks, amount of main memory, and interconnection network bandwidth. We model the performance of the algorithms analytically to determine elapsed time, resource utilization, and other quantities as functions of the workload and architectural parameters. The join algorithms overlap computation, disk transfers, and interconnection network transfers. The analysis models this overlap and identifies bottlenecks that limit the algorithms' performance. We do not model multiple simultaneous join operations, and therefore do not compute system throughput. Based on this analysis, we answer the following questions: (1) How do the algorithms compare in performance? When does one outperform another? (2) How does response time vary as a function of the architectural parameters? (3) How does response time vary with the workload? (4) Does shared memory help algorithm performance? To what extent? (5) What are the architectural bottlenecks? How could they be alleviated? Based on the above study, we draw several conclusions regarding parallel processing of the join operation.

Chapter 7 presents the results of our investigation of fault tolerance in very large database systems. A very large database is usually heavily used and many users and applications depend on it. Downtime or unavailability of such a system is expensive and affects critical applications dependent on it. We studied the effect of fault tolerance techniques and system design on system availability. Specifically, we attempted to answer the following questions: What are the main parameters that affect fault tolerance of a very large database system? How do you evaluate their effect on fault tolerance? How important are various fault tolerance techniques? What are the trade-offs that should be considered when designing a very large database system with a desired degree of fault tolerance? A generic multiprocessor architecture is used that can be configured in different ways to study the effect of system architectures. Important parameters studied are different system architectures and hardware fault tolerance

techniques, mean time to failure of basic components, database size and distribution, interconnect capacity, etc. Quantitative analysis compares the relative effect of different parameter values. Results show that the effect of different parameter values on system availability can be very significant. System architecture, use of hardware fault tolerance (particularly mirroring) and data storage methods emerge as very important parameters under the control of a system designer.

During Phase II, the results of the six investigation studies were combined to develop a methodology for specifying high performance, highly available D/KBMS architectures for very large data/knowledge base (D/KB) environments. Chapter 8 describes this methodology, which is described as a set of policies and steps. The policies are meant to serve as a guide to the D/KBMS designer in making appropriate decisions for the following critical D/KBMS design issues: overall D/KBMS functionality, knowledge representation, rule storage, D/KB query processing, D/KB update processing, D/KBMS functional partitioning, least fixed point evaluation, join processing, D/KBMS hardware architecture, and fault tolerance. The steps are presented as a recipe for D/KB query and update processing.

Briefly, using this methodology, we would design a high performance D/KBMS by first designing a parallel relational database machine that employs the parallel and pipelined join algorithms described in chapter 6. Next, we would design parallel algorithms for LFP evaluation using the above join strategies, data flow and pipelining techniques, and semi-naive evaluation. Finally, we would design a compiler that compiles Horn clause queries to relational algebra augmented with a general least fixed point operator and that uses the generalized magic sets strategy for restricting the search space to the relevant base relation tuples.

Chapter 9 describes a data/knowledge base management testbed that we designed and implemented on top of a commercial relational database system. The testbed is intended to serve as both a demonstration and performance measurement and evaluation platform. As a demonstration platform, the testbed illustrates the motivation and basic functionality of a D/KBMS, the components of a D/KBMS architecture, alternative implementations of these components and their relative tradeoffs, and the factors contributing to D/KB query compilation and execution time. As a performance measurement and evaluation platform, the testbed allows us to make quantitative performance measurements and to study system performance sensitivity and behavior with respect to several parameters.

Chapter 10 describes the VLPDF demonstration plan. The demonstration consists of three experiments designed to demonstrate the motivation and functionality of a D/KBMS and the components of a D/KBMS architecture.

Chapter 11 describes several experiments designed to quantitatively measure D/KBMS performance and to understand D/KBMS performance sensitivity and behavior with respect to various system parameters. The basic motivation for doing these experiments is to justify the D/KBMS architecture specification methodology described in chapter 8. That is, to show that this methodology can indeed be used to design high performance D/KBMSs. The chapter describes D/KBMS performance

measures, system parameters affecting these measures, test results, analysis of these results, and the conclusions drawn from them.

Chapter 12 summarizes the key conclusions from this work and indicates several directions for future work.

CHAPTER 2

Application Interface

The choice of an application interface language for a D/KBMS is a difficult one. No existing language is uniformly superior to all others with respect to our requirements. We have chosen PARLOG, a parallel logic-based language developed by Clark and Gregory [Clar86]. We believe that PARLOG represents the best choice among existing languages. Its principal weaknesses are limited data structuring capabilities and possible inefficiencies in implementing object-oriented knowledge representation. However, it is superior to other languages in many respects. In particular, it permits a high degree of parallelism in both procedural computation and database access.

The chapter is organized as follows. Section 2.1 lists our requirements for an application interface language. Section 2.2 discusses the three language classes considered: imperative, functional, and logic, and justifies our preference for logic-based languages. Section 2.3 compares alternative logic-based languages and justifies our choice of PARLOG. Section 2.4 reviews PARLOG with respect to the original requirements. Section 2.5 describes PARLOG in some detail.

2.1. Language Requirements

We have identified three major requirements for the application interface language. The language must

- be amenable to large-scale parallelism,

- be a suitable base for implementing an expert system shell, and

- support efficient nonprocedural database access.

The following sections explain these requirements.

2.1.1. Large-Scale Parallelism

The language must be amenable to parallel execution. Here, we are looking for large-scale parallelism, where the degree of parallelism possible is proportional to the volume of data being processed. Such parallelism is currently exploited in multiprocessor relational database machines such as the Teradata DBC-1012 [Tera83]. A language that permits only small-scale parallelism, e.g. pipelined execution, is of less interest. We believe that large-scale parallelism is essential to future C^3I applications. It must be possible to scale the D/KBMS architecture to the size of the problem being tackled; applications written in the language should not require modification to take advantage of an expanded D/KBMS system configuration.

2.1.2. Base for Expert System Shell

Advanced C^3I information management applications must perform many of the knowledge-based activities associated with expert systems: interpretation, diagnosis, monitoring, prediction, and planning [Stef82]. Therefore we expect that these applications will require the full range of facilities provided by expert system shells such as KEE, ART, and LOOPS. These facilities include:

> forward chaining (data-driven reasoning)
> backward chaining (goal-driven reasoning)
> procedural computation
> object-oriented knowledge representation
> evidential reasoning
> models of time and hypothetical worlds
> belief maintenance
> nonmonotonic reasoning
> explanation facilities

Forward chaining is a rule-based inference mechanism in which the rules are used in a forward direction, from prerequisites to conclusion or action, to derive new information or a new state from existing information or the current state. OPS5 [Forg81] is an example of a forward chaining language. Forward chaining is also called *data-driven* reasoning because it accepts new information and derives all consequences of this information. In the C^3I domain, forward chaining is useful in applications that monitor the current state of the world for significant events that may require a response.

Backward chaining is a rule-based mechanism in which rules are used backwards, from conclusion or action to prerequisites, to determine whether a given statement or goal can be supported by existing information. Backward chaining is also called *goal-driven* reasoning. Prolog [Cloc84] is the best-known backward chaining language.

Procedural computation is prescriptive computation of results from arguments, as found in languages such as Pascal and LISP.

Object-oriented knowledge representation models real world objects and concepts as objects in the Smalltalk sense, i.e., combinations of data and procedures to operate on the data. Frames [Barr81] are a form of object-oriented knowledge representation. An object's data can be accessed only via its procedures, which are invoked by "sending the object a message." Different objects can respond to the same message in their own way, according to their associated procedures. Objects can be combined into larger objects in an aggregation hierarchy. An inheritance hierarchy can also be defined among objects, so that an object can inherit by default the properties (procedures or data) of an object higher on the hierarchy. An object-oriented knowledge representation provides a concise way of modeling many real world situations.

Evidential reasoning is a form of reasoning in which hypotheses have associated probabilities or certainty factors. Mycin [Shor76], an expert system for diagnosing and treating infectious diseases, employs probabilistic reasoning. Evidential reasoning can be applied to diagnostic and planning tasks in the C^3I domain as well.

Models of time and *hypothetical worlds* refer to the ability to model not just the current state of the world, but the sequence of states that constitutes the past, and possibly one or more hypothetical sequences of future states. Each state in the sequence is an incremental modification of the previous one, typically representing the addition of a new fact or assumption. Hypothetical worlds can be used to explore alternative strategies in a planning activity.

Belief maintenance is a facility that can be built into a logic-based inference system to record dependencies among propositions, detect inconsistencies among them, and permit retraction of propositions.

Nonmonotonic reasoning is a reasoning system in which a default assumption about an object or situation can be made in the absence of specific information. This assumption can later be overridden when additional information becomes available. Retraction of the assumption requires retraction of any conclusions deduced from the assumption, using some form of belief maintenance.

An *explanation facility* is a mechanism by which an inference system displays how it proved or failed to prove a particular conclusion, tracing the conclusion back via inference rules to base facts.

The application interface language need not support these facilities directly. However, it should be possible to implement them reasonably efficiently in the language.

2.1.3. Support for Nonprocedural Database Access

Most expert systems today use a relatively small amount of data that can be stored entirely in main memory. Advanced C^3I applications will have large data and knowledge bases, necessitating an underlying database management system. Therefore, the language must support nonprocedural retrieval and update of stored data, as in a relational DBMS, and must permit efficient execution of traditional relational queries.

2.2. Alternative Language Classes

The candidate languages can be divided into three major classes: imperative, functional, and logic. The imperative languages are characterized by sequences of commands or statements that make incremental changes to a global program state contained in a set of variables. Examples of imperative languages are traditional programming languages such as FORTRAN and Pascal, and more modern object-based languages such as Smalltalk and CLU.

Programs in functional languages are essentially definitions and applications of functions. There is no notion of operations on named objects, and therefore there are no side effects. Examples of functional languages include pure LISP, FP, and dataflow languages such as Val and Id. LISP, as it is generally used today in expert systems and other applications, is an imperative rather than a functional language; great use is made of side effects.

Programs in most logic programming languages are composed of Horn clauses, which have the form

head :− *body*.

where *head* is zero or one atomic formulas (predicates with arguments supplied) and *body* is a conjunction of zero or more atomic formulas. All arguments that are variables are implicitly universally quantified. The logical interpretation of a Horn clause is that the body implies the head. For example, the Horn clause

$a(X,Y) :- b(X,Z), c(Z,Y).$

means that for all X, Y, and Z, $b(X,Z)$ and $c(Z,Y)$ implies $a(X,Y)$. (We use the Prolog convention that constants and predicate names begin with lower case letters, while variables begin with capital letters.)

An empty Horn clause body is considered true. Therefore a Horn clause with an empty body states that the head is always true. Such a Horn clause is called a *fact*. Facts can be written with no implication sign, e.g.,

parent(a,b).

means that a is the parent of b.

An empty Horn clause head is considered false. Therefore a Horn clause with an empty head states that the conjunction of atomic formulas in the clause's body is false. This refutation of the body can be used to initiate a resolution-based proof that the body is in fact true [Cloc84]. In the course of this proof, all variable instantiations that make the body true can be discovered. A Horn clause with no head is therefore called a *goal* or *query*.

Horn clauses do have limited expressiveness: it is difficult to express indefinite information such as "object X is either an airplane or a missile." However, permitting the expression of indefinite information makes proof procedures much less efficient.

Logic languages also give a procedural interpretation to a Horn clause: if the head is a goal, invoke the atomic formulas of the body as subgoals. Logic languages differ from functional languages in that the predicates can (generally) be used in more than one direction. The following are all acceptable goals:

:− *parent*(a,b).

:− *parent*(X,b).

:− *parent*(a,Y).

:− *parent*(X,Y).

Predicates that can be used in more than one direction are called *multi-use* programs. In contrast, imperative and functional languages compute in only one direction, from argument to result.

Each language class is superior to each other class with respect to at least one of our requirements. Imperative languages, particularly LISP, form the basis of current expert system shells, and therefore demonstrably support the facilities listed earlier. Expert system shells based on logic or functional languages, e.g. APES [Hamm82], are not nearly as highly developed. Furthermore, many imperative languages have been

extended with interfaces to database management systems. The differences between the common imperative languages and relational data manipulation languages make these interfaces awkward and inefficient, however.

Current imperative languages discourage exploitation of parallelism. They typically require explicit specification of concurrent processes, and hence make it difficult to achieve parallelism proportional to the amount of data being processed. In contrast, parallelism is often implicit in functional and logic languages. Furthermore, imperative languages often permit uncontrolled side effects that preclude partitioning a problem into independent subcomputations. We view exploitation of parallelism as the essence of the research to be done. Therefore, we have excluded imperative languages from consideration as the parallel inference language.

This is not to say that large-scale parallelism cannot be achieved using imperative languages. Object-oriented principles can be used to limit side effects and therefore promote parallelism. Parallel object-oriented languages are a current research area.

Functional languages have implicit parallelism in the concurrent evaluation of arguments to a function. The U-Interpreter for the Id language also exhibits large-scale parallelism [Arvi82]. However, the lack of side effects in functional languages makes it impossible to do such basic operations as update a database. For example, the FQL language [Bune82] is a functional language designed for querying databases. It models stored data as functions, and has a syntax similar to FP. However, it has no update capability. For this reason, we have eliminated functional languages as unsuitable to database-oriented applications.

This leaves logic languages. Logic languages provide both nonprocedural database access and the opportunity for large-scale parallelism.

A logic language can provide the functionality of a relational database system in the following way. Relations can be represented as sets of facts. Queries are simply goals. Views (derived relations) can be defined using ordinary Horn clauses, as illustrated below. Updates can be accomplished using built-in predicates that have the side effect of adding or deleting clauses. It is easy to show that a logic language with the semantics defined above is relationally complete.

Three forms of parallelism have been identified for logic languages: *and-parallelism, stream parallelism,* and *or-parallelism.* And-parallelism is the parallel resolution of different literals in the body of a Horn clause. This re $\cdot$ lution must be coordinated when the literals share logic variables. Stream parallelism is a form of and-parallelism involving parallel binding and use of a structured value, typically a list. Or-parallelism is parallel pursuit of alternative proofs of a literal.

And- and or-parallelism have analogs in relational database terms. Consider a database with relations

$emp(Eno, Ename, Date\_of\_birth, Dno)$

$dept(Dno, Dname, Location)$

The SQL query to find the names of all employees in the Sales department is

SELECT Ename FROM emp, dept
WHERE emp.Dno = dept.Dno AND dept.Dname = "SALES"

A relational algebra tree to compute this query is shown in Figure 2.1.

SELECT ENAME FROM EMP, DEPT

WHERE EMP. DNO = DEPT. DNO AND DEPT. DNAME = 'SALES'
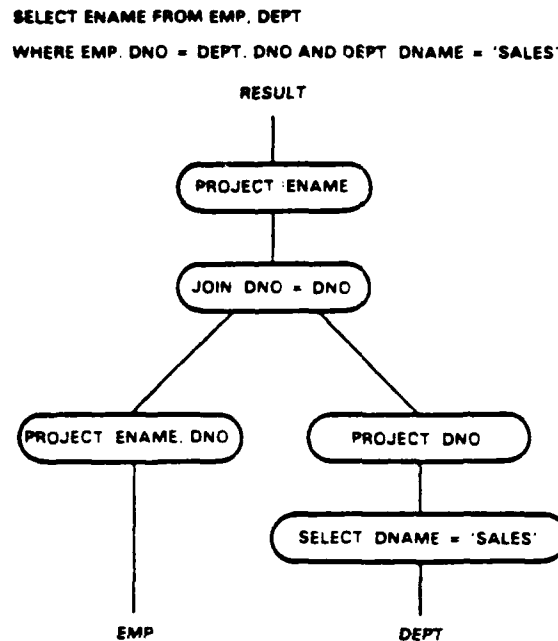
Figure 2.1. Relational Algebra Tree

The equivalent query in a logic language would be

$$emp\_dept(Eno, Ename, Dname) := emp(Eno, Ename, \_, Dno),$$
$$dept(Dno, Dname, \_).$$

$$:- emp\_dept(\_, emp, 'SALES').$$

(Here, we have defined the *emp_dept* view before issuing the query.) And-parallelism in the resolution of the first clause's body is realized by concurrent evaluation of the two independent branches of the relational algebra tree, with the join operation implementing the coordination of the shared logic variable *Dno*. Or-parallelism is realized by processing multiple tuples at a time in the select, project, and join operations. Stream parallelism appears to have no analog in relational database systems because variables are bound to unstructured values such as numbers and strings, not lists or other structures that could be used as they are generated.

2.3. Alternative Logic Languages

We decided to concentrate on logic languages because they provide nonprocedural database access and the potential for large-scale parallelism. We considered three logic languages: Prolog, Concurrent Prolog [Shap83], and PARLOG.

The semantics of Prolog are defined in terms of a sequential execution model. In this model, the order of the clauses in a Prolog "database" is significant; the database is scanned sequentially from top to bottom when attempting to satisfy a goal. Within a clause, the atomic formulas in the body are satisfied left to right in order. Finally, the cut operator, when executed, prevents searching for later clauses to satisfy the goal that the current clause is attempting to satisfy.

This top-down, left-right execution model does not exploit and-, or-, or stream parallelism. Referring to the *emp_dept* query above, the top-down search rule prevents an or-parallel search of the dept tuples to find the "SALES" department. It also prevents an and-parallel execution of the projection of *emp* and selection of *dept*, should this be the chosen evaluation strategy.

The Prolog execution model dictates a particularly inefficient execution of queries involving joins, such as the emp_dept query shown above. The join between the emp and dept relations is executed essentially as a nested loop join with emp as the outer relation and dept as the inner relation. The execution time of this join is proportional to the product of the relation sizes. Joins can often be performed much more efficiently using a sort-merge or hash-based algorithm [Vald84]. The inefficiency of the nested loop join relative to other algorithms increases with the size of the relations. Even if a nested loop join is to be used, using the smaller relation (presumably dept in this case) as the outer relation results in faster execution when the relation tuples are stored on disk.

It is possible to analyze Prolog programs to detect instances where a clever execution strategy, such as an efficient join or parallel execution of clauses, produces the same result as the standard execution model. An optimizing Prolog compiler could perform this analysis and choose the clever strategy where possible. However, we believe that optimizing a sequential language for parallel execution is inferior to starting with a language designed for parallel execution in the first place, assuming that such a language is available.

A further problem with Prolog is that it does not always provide the natural fixed point semantics for execution of a set of Horn clauses. This occurs when the Horn clauses are recursive. For instance, consider the following definition of the *ancestor* relation:

$$ancestor(X,Y) :- parent(X,Y).$$

$$ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).$$

The definition of ancestor is left-recursive; application of Prolog execution rules leads to an infinite loop. This is a problem with Prolog semantics and not with any implementation of those semantics: a clever execution strategy does not alleviate the problem.

Having eliminated Prolog, we examined two variants of Prolog designed for parallel execution: Concurrent Prolog and PARLOG. Concurrent Prolog presents a third interpretation of logic programs in addition to the declarative and procedural interpretations: atomic formulas can be executed as processes. A Horn clause represents an expansion of a process (the predicate in the consequent) into a set of processes (the predicates in the body.) Processes communicate with each other via shared logic variables. This is clear and-parallelism. Stream parallelism is realized when one process binds a logic variable to a structured value (typically a list), producing the structure as another process consumes it. A synchronization mechanism called *read-only variables* is used to delay the consumer process when it attempts to reference a variable that the producer process has not yet bound. These concepts appeared in an earlier logic language, the "Relational Language" [Clar81].

In this interpretation of logic programs, there is no generation of alternative proofs for a goal via or-parallelism or backtracking. Concurrent Prolog checks multiple clauses in parallel for applicability, and then nondeterministically chooses one of them. A common example is nondeterministic merge [Shap83]. The merge relation interleaves its first two arguments nondeterministically to produce the third argument:

$$merge([X \mid Xs], Ys, [X \cdot Zs]) :- merge(Xs?, Ys?, Zs).$$

$$merge(Xs, [Y \mid Ys], [Y \cdot Zs]) :- merge(Xs?, Ys?, Zs).$$

$$merge(Xs, [], Xs).$$

$$merge([], Ys, Ys).$$

(The question marks indicate read-only variables.) Concurrent Prolog has no back-tracking: once the choice is made between the first two clauses, the choice is never revisited. The merge relation produces *some* merging of its input arguments, not *all* of ʰ Thus, the semantics of Concurrent Prolog, while appropriate for the construction oncurrent systems, are quite different from fixpoint semantics and therefore inappropria·ᵉ for database retrieval.

PARLOG, in contrast, defines two different kinds of relations (predicates): *single-solution relations* and *all-solutions relations*. Single-solution relations are executed using the parallel process semantics similar to those of Concurrent Prolog. However, their semantics have been defined to eliminate runtime management of multiple binding environments. Each single-solution relation must have a *mode declaration* that indicates which of the relation's arguments are inputs and which are outputs. For example, the following relation computes the sum of the elements on a list:

$$mode\ sum(List?, Sum\hat{}).$$

$$sum([], 0).$$

$$sum([X \mid Xs], S1) :- sum(Xs, S),\ S1\ is\ S + X.$$

Single solution relations are thus basically functions that compute results from arguments. They provide a procedural computation facility similar to LISP. It is possible, though awkward, to specify single solution relations that compute in more than one

direction. (PARLOG does provide nondeterminism and logic variables, both of which are absent in LISP.)

All-solutions relations are defined using pure Horn clauses that can be executed with fixpoint semantics. All solutions relations have no mode declarations, and can provide nonprocedural database access as illustrated earlier.

The interface between the two kinds of relations is provided by the "set" constructor, which constructs a list of the results of an all-solutions relation query. This list can then be manipulated by single-solution relations. For example, the following clauses computes the sum of the ages of Sam's children.

$sum(Ages, Number)$.

$set(Ages, A, (parent(Sam, X), age(X,A)))$.

Here, $Ages$ is the list generated by the set constructor. It is a list of terms A, where A must satisfy $parent(Sam,X)$ and $age(X,A)$ for some child X. The combination of capabilities provided by single- and all-solutions relations makes PARLOG attractive for complete applications. Section 2.6 gives a more complete description of PARLOG.

2.4. PARLOG and the Language Requirements

Let us review PARLOG with respect to the language requirements stated earlier.

2.4.1. Large-Scale Parallelism

Large-scale parallelism in single-solution relations is hindered by the lack of an array or similar direct-access data structure in PARLOG. (Other logic languages have the same limitation.) Data that might be organized as an array in a conventional language must be organized as a list (or perhaps as a tree, for faster access) in a logic language. In order to perform some operation on all elements of a list, the list must be traversed element by element, spawning a process to perform the desired operation on each element, as in the sum example above. This list traversal time, which is linear in the number of elements, may dominate the total processing time. Organizing the data as a tree can reduce the traversal time to be logarithmic in the amount of data, assuming enough processors.

Large-scale parallelism can be achieved in all-solutions relations using or-parallelism in a multiprocessor architecture. For instance, the Teradata database machine partitions the tuples of each relation across an array of processor/disk pairs [Tera83]. The common relational operations can then be executed in parallel in these partitions. An implementation of all-solutions relations could make use of similar techniques.

2.4.2. Base for Expert System Shell

Forward chaining: Existing logic-based languages, including PARLOG, have no built-in forward chaining facility. However, forward chaining can be implemented on top of logic languages [Subr85].

Backward chaining: PARLOG, like Prolog, provides backward chaining for goal-directed reasoning. When multiple alternatives must be explored to find a proof for a goal (the usual case), all-solutions relations should be used.

Procedural computation: PARLOG's single-solution relations can be used as functions, since each argument of a relation must be declared as input or output. These declarations permit more efficient implementation of procedural computation than would be the possible in Prolog. However, PARLOG lacks structured data types, such as arrays and records, that are found in conventional programming languages, though they could be added. In this respect, PARLOG is probably no worse than early dialects of LISP. Each of these features can be mimicked in PARLOG. For instance, arrays can be implemented (inefficiently) using relations. However, the lack of direct support makes PARLOG a poor vehicle for applications that make heavy use of them. To overcome this problem, an interface between PARLOG and the C language has already been developed to support multiprocessing for numeric applications [Butl85].

Object-oriented knowledge representation: Shapiro and Takeuchi have shown that object-oriented programming can be performed in Concurrent Prolog; similar techniques work for PARLOG, using single-solution relations. Each object is represented by a perpetual process that receives a stream of messages as input and generates a stream of messages as output. The state of the object is maintained in logic variables local to the process. While this implementation may be functionally adequate, the cost of dedicating a process to each object may be excessive. The execution model for PARLOG single-solution relations presented in the next chapter is designed to minimize the overhead of process creation and termination. It may also be possible to develop compile-time techniques to reduce this overhead.

Evidential reasoning, belief maintenance, nonmonotonic reasoning, and *explanation facility:* PARLOG provides no direct support for these facilities. However, their implementation in PARLOG appears relatively straightforward [Subr85].

2.4.3. Support for Nonprocedural Database Access

As stated earlier, PARLOG provides a relationally complete nonprocedural language using its all-solutions relations. It goes beyond relational completeness by providing recursive queries. Aggregate functions can be implemented easily using a combination of single- and all-solutions relations, as illustrated in the program for summing the ages of Sam's children.

To summarize, PARLOG is not uniformly superior to all other languages with respect to our requirements. In many cases, such as procedural computation and object-oriented knowledge representation, other languages are clearly superior to PARLOG. However, we are not aware of another language that meets the entire set of requirements better than PARLOG. Parallel languages for symbolic computation are an active research area, and we expect better languages to emerge in the next few years.

2.5. An Overview of PARLOG

In this section, we provide an overview of PARLOG. For a more detailed exposition, the reader is referred to [Greg85], and [Clar86]. As mentioned in section 2.3, PARLOG features two kinds of relations: single-solution relations, and all-solutions relations. All-solutions relations can appear in the body of single-solution relations. The evaluation semantics for these relations are completely different. Indeed, they can be viewed as two different languages — all-solutions relations constituting a query language and single-solution relations a parallel applications programming language. An all-solutions relation query can be evaluated by computing the least fixed point of the Horn clauses defining this relation. Thus, PARLOG can be considered as having a Horn clause query language embedded within it.

Single-solution relations compute just a single solution to a query. They provide a procedural computation facility similar, but not identical, to that provided by functional programming languages. The difference arises due to the *logical variable,* which is basically a variable in a logic programming language. However, terms bound to such variables may be only partially instantiated, i.e., they may contain variables. If such terms appear in input argument positions of relation calls, the call may bind the uninstantiated variables. This is in contrast to functional programming languages, where arguments of a function call are fully instantiated. Also, in functional programming languages, evaluation of a function cannot produce partially instantiated data structures. All-solutions relations compute all the solutions to a query. Therefore, they are suitable for non-procedural access to databases. We describe the two halves of the language below.

2.5.1. Single-solution Relations

A single-solution relation consists of a *mode declaration,* and a set of *guarded clauses.* A mode declaration identifies arguments of a relation as being inputs or outputs. For example, the mode declaration $R(?, \hat{\ })$ specifies that the first argument of relation R is an input, while its second argument is an output.

A guarded clause is a clause of the form:

head — guard : body

where *head* is a literal, and where *guard* and *body* are possibly empty conjunctions of *literals.* If the guard is empty, the operator ":" is not present. A literal is a tuple prefixed by a relation name. An example of a guarded clause is the following:

$$R(t_1, \ \cdots \ , t_n) - G_1(p_1, \ \cdots \ p_k), \ \cdots \ G_m(q_1, \ \cdots \ q_l) :$$
$$B_1(r_1, \ \cdots \ r_a), \ \cdots \ B_p(w_1, \ \cdots \ w_h)$$

PARLOG features both, *sequential* and *parallel* conjunctions. An example is the following.

$$(R_1 \ \& \ (R_2 \ , \ R_3)), \ R_4, \ R_5$$

The sequential conjunction operator "&" indicates that literals following it are to be

evaluated only after all previous literals have succeeded. On the other hand, the parallel conjunction operator "," indicates that a "," separated group of literals may be evaluated in parallel. Thus, in the example above, R_2 and R_3 may be evaluated in parallel, but only after R_1 has been successfully evaluated. Thus, the "&" and "," operators have a control significance—they dictate the order in which literals in a conjunction are to be evaluated.

Mode declarations impose a directionality on logic programs. If a particular argument position has a mode annotation "?", then a non-variable term, say $[a \mid b]$, appearing in that argument position in the head of a clause can be used only for *input matching*. That is, the call argument for that position should be a substitution instance of $[a \mid b]$.

Similarly, if a particular argument position has a mode annotation "^", then the call argument for that position must an uninstantiated variable, else we have run-time error. A non-variable term appearing in that argument position in the head of clause can be used only for *output matching*.

The principal advantage of mode declarations is that most of the unification in PARLOG can be compiled. The details are explained in [Greg85].

If a call argument is instantiated enough to be able to determine that it is not a substitution instance of a non-variable term appearing in an input argument position in the head of a clause, the attempt to use that clause is aborted. However, if the call argument is not yet instantiated enough to be able to make a decision, the attempt to use the clause is suspended.

A clause is called a *candidate clause* for a relation call if the head of the clause input matches with the call on all the input arguments, and if it has a successfully terminating guard. A clause is called a *non-candidate clause* if either or both these conditions are false. As described before, input matching may cause suspension. In this case, the clause cannot yet be classified as a non-candidate clause.

The set of clauses defining a relation may be separated either by a ";" operator, or a "." operator. For example, a relation R may be defined by a set of five clauses, composed like so:

$$(C_1; (C_2 \cdot C_3)) \cdot C_4 \cdot C_5$$

The ";" and "." operators control the order in which the set of clauses are to be tried in order to find a candidate clause. A ";" indicates that clauses following it are to be tried only if all of the preceding clauses prove to be non-candidate clauses. However, all clauses in a "." separated group of clauses may be tried in parallel. Thus, in the example above, the clauses C_2 and C_3 may be tried in parallel, but only after C_1 proves to be a non-candidate clause. Thus, just like the "&" and "," operators, the ";" and "." operators have a control significance—they dictate *how* a solution to a query is to be found.

The evaluation of a relation call starts out with an attempt to find a candidate clause, in the order specified by the ";" and "." operators. The implementation is free

to choose any candidate clause. When it picks a candidate clause, it is said to *commit* to that clause, i.e. the body of this clause is evaluated to find one solution to the call. There is no backtracking on this choice. This is referred to as *committed choice non-determinism*. The operator, ":", separating the guard and body of a guarded clause, is referred to as the commit operator.

Guards of clauses being tried during the search for a candidate clause, are not allowed to bind variables in the call until commitment. Guards that don't bind variables in the call are called *safe guards*. PARLOG enjoys the advantage that the safety of guards can be checked at compile time—a property that is significantly conducive to efficient implementation. The compile time safety check obviates the need to maintain, at run-time, multiple environments for the different guard evaluations and export them upon commitment, as is the case in Concurrent Prolog [Shap83].

Binding of call variables is made, in the form of output matching, upon commitment. Since there is no backtracking on the choice of the candidate clause, bindings made to variables never need be retracted. Thus, variables in PARLOG have the *single-assignment* property.

We now describe the distinguishing attribute of single-solution relations, viz., stream-*AND* parallelism. This form of parallelism arises when there are multiple relation calls working concurrently on evaluating the same solution. They communicate by passing bindings through shared variables. It is to be contrasted with all-solutions *AND* parallelism, which arises when multiple solutions to a query are available. Stream-*AND* parallelism requires that no more than one solution to a call be computed. If such is the case, the solution can be generated incrementally, via a series of approximations. If an approximate solution is never retracted, it can be communicated immediately to other calls, thus making stream-*AND* parallelism easy to implement. Single-solution relations in PARLOG feature this form of parallelism since shared variables in such relations have the single-assignment property, and since only one solution to such relations is computed (due to committed choice non-determinism). Stream-*AND* parallelism is typically used in the construction of a list—different portions of it would be instantiated by different relation calls working concurrently.

Object oriented programming is possible using single-solution PARLOG relations. This is because, unlike functions, such relations can bind variables in their input argument terms. Whenever a single-solution relation does this, a *back communication* to the calling relation occurs. This mechanism is the basis for object-oriented programming in PARLOG: objects are implemented as relations, a message is a partially bound term given as input to a relation, and a reply is sent by completing the term binding.

Like other logic programming languages, PARLOG features the *metacall*, which allows data to be executed as programs.

Any PARLOG program can be automatically translated into a program in a lower level language, called Kernel-PARLOG. It is the first step in the compilation of PARLOG programs. See [Greg85] for the details. We have used Kernel-PARLOG as the basis of our work in parallel inference architectures.

Kernel-PARLOG does not have any mode declarations. Instead, the mode declarations in PARLOG programs, are used to compile input and output matches to explicit one way unification ($<=$) calls. In the input match call, $nt <= v$, the left argument is a non-variable term, and the right argument is a variable. During evaluation of the call, variables in nt are bound, so as to make nt and v syntactically identical. The call suspends if it can proceed only when variables in v get instantiated. In the output match call, $v <= nt$, v must be an uninstantiated variable at the time of the call, in which case it is bound to nt. Otherwise, there is a run-time error.

2.5.2. All-Solutions Relations

As the name implies, all-solutions relations compute all solutions to a query. In fact they compute a list of all the solutions. This list may be consumed by a single-solution relation call. As discussed in section 2.3, the set constructor is the interface between single-solutions relations and all-solutions relations. The operational semantics of the set constructor are not specified. Thus, the PARLOG programmer may make no assumption about the order in which the solutions are computed. This allows considerable flexibility in the implementation of the set constructor. One possibility is a Prolog-style backtracking evaluation. Another possibility is to have sets of solutions computed independently, and then doing a join operation on them.

2.6. Conclusion

In this chapter, we have presented our investigation of the alternatives for the application interface language for a D/KBMS. This language forms the basis for developing C^3I applications such as planning, monitoring, threat assessment, interpretation, etc. We identified three major requirements for this language. It must be amenable to large scale parallelism, be a suitable base for implementing an expert system shell, and support efficient nonprocedural database access. Among the various imperative, logic, and functional languages that we evaluated, PARLOG best met these requirements. The next chapter describes our investigation of parallel architectures for executing PARLOG.

CHAPTER 3

Parallel Architectures for the D/KBMS Application Interface

In this chapter, we describe our investigation of parallel architectures for executing PARLOG, our choice for the D/KBMS application interface language. Our approach to this problem was the following:

- Design a parallel abstract machine for PARLOG. The abstract machine is not a physical hardware architecture. Its purpose is to lay bare all the functions that need to be performed to execute PARLOG programs as well as the data objects created and manipulated by these functions.

- Develop a simulator for the parallel abstract machine. The simulator serves as an implementation for the language. The simulator basically implements the functions and the interactions between them. It serves as a tool for collecting data on the execution behavior of the language, and analyzing this data.

- Analyze the run-time execution behavior of PARLOG programs using the simulator. The analysis focuses on questions such as: How much parallelism is there? What is the granularity of the parallelism? What are the communication patterns? What operations are performed most frequently? Do they require hardware support? If so, what kind? etc.

- Design a suitable parallel hardware architecture (both interconnection network and processing element) for implementing the functions, and map the abstract machine (i.e., the functions) onto it.

This approach was motivated by the fact that while the area of parallel architectures for concurrent logic programming languages enjoys vigorous activity (see [Ito85, Mill84, Greg85, Hali84]), it is still very much a research area and that much work still needs to be done. In particular, the run-time execution behavior of concurrent logic programs needs to be studied. We believe that the chances of obtaining high performance are greatly enhanced if architectural decisions are based on an analysis of the run-time execution behavior of programs. Such an analysis will ensure that the architecture is well matched to the language semantics, which in turn, is very important for high performance.

Carrying out all the above steps is a major effort in itself. However, as part of the VLPDF effort, we completed the first two steps and briefly investigated the feasibility of using the Connection Machine for executing PARLOG by studying the mapping of the abstract machine onto the Connection Machine.

This chapter presents the results of the above work and is organized as follows. Section 3.1 describes a parallel computational model for PARLOG to motivate the design of the abstract machine. Section 3.2 describes the abstract machine. Section 3.3

describes our investigation of the feasibility of mapping this abstract machine onto the Connection Machine architecture. Section 3.4 summarizes our conclusions from this part of the VLPDF investigations.

3.1. A Parallel Computational Model for PARLOG

In this section, we describe a parallel computational model for PARLOG in order to motivate the design of the abstract machine. Specifically, we

- describe the data objects created and manipulated during execution of PARLOG programs,
- describe how PARLOG programs are represented, and
- describe what operations need to be performed in order to execute such programs.

The description of data objects includes:

- a description of the different types of data, both scalar and structured, featured in PARLOG;
- a description of how data objects are represented;
- a description of how they will be addressed—globally addressed in a shared memory system, or locally addressed in a loosely-coupled system; and
- a description of how data objects are aggregated to form ever more complex objects.

The description of operations includes:

- a description of the PARLOG control structures,
- a description of how these operations are scheduled for execution,
- a description of how these operations are synchronized, and
- a description of how data objects are created.

3.1.1. PARLOG Control Structures

There are basically four control structures in PARLOG: *sequential conjunction, parallel conjunction, sequential search,* and *parallel search.* The abstract *AND/OR* process model for PARLOG proposed by Gregory in chapter 6 of his dissertation [Greg85] is an ideal vehicle for understanding the control structures of PARLOG. It overcomes the weakness of the *AND/OR tree* representation—a graphical representation that captures the different evaluation paths arising during evaluation of a query in a Horn clause program—viz., lack of control information, i.e., information about *how* a solution to a query is found.

In this model, a process is created for: evaluating user-defined literals, non-compilable primitives, and conjunctions of literals; and for searching for a candidate clause during evaluation of a literal. The state of a PARLOG evaluation is represented by a process structure called the *AND/OR process tree.* The nodes in this tree are processes. The leaf processes are either runnable or suspended on some variable. The non-leaf processes are not runnable. They await results from their child processes. There

are two types of non-leaf processes: *AND* processes and *OR* processes. A process assumes a type *AND* if it is to evaluate a conjunction of literals. A conjunction can be a sequential conjunction or a parallel conjunction. A conjunction may consist of just a single literal. A process assumes a type *OR* if it is to search for a candidate clause among the clauses defining a relation.

The evaluation of a literal R starts out by searching for a candidate clause. If a sequential group of clauses, i.e., ";" separated clauses, is to be searched, the process evaluating the literal initially spawns a child process to evaluate the guard of the first clause. It then becomes an OR process. Next, it sets its *continuation*, which for an *OR* process is the code to be executed if the guard that is currently being evaluated fails. The guards of the other clauses are also evaluated in the same child process. However, for the other clauses, a guard is evaluated only if the guard of the preceding clause fails. For example, if a relation R is defined as follows:

$$R - G_1 : B_1;$$
$$R - G_2 : B_2;$$
$$R - B_3.$$

the process evaluating the literal R spawns a process to evaluate G_1. It then becomes an *OR* process. Finally, it sets its continuation to the code that will spawn a process for evaluating G_2.

If a parallel group of clauses, i.e., "." separated clauses, is to be searched, the process evaluating the literal spawns child processes for evaluating the guards of these clauses. These processes are executed in parallel. The spawning process becomes an *OR* process. It then sets its continuation to the code to execute in case all the spawned processes fail. For example, if a relation R is defined as follows:

$$R - G_1 : B_1.$$
$$R - G_2 : B_2;$$
$$R - B_3.$$

the process evaluating R spawns two processes, one for evaluating the guard G_1, and the other for evaluating the guard G_2. These two processes are evaluated in parallel. The continuation of the spawning process is then set to the code that will spawn a process for evaluating the body B_3.

When a process evaluating a guard succeeds, its parent will *commit* to the clause containing the guard, in case it hasn't committed to any other clause. The action of committment is manifested by the parent process proceeding to evaluate the body of the clause committed to, and terminating the processes evaluating the other guards, if any. Thus, in the last example above, if G_2 succeeds before G_1, the parent process will proceed by evaluating B_2, and terminating G_1.

If a sequential conjunction of literals, i.e., "&" separated literals, is to be evaluated, the process evaluating the conjunction checks if the conjunction begins with a sequence

of primitive instructions (see chapter 5 of [Greg85]). If so, the primitive instructions are executed *within* the process evaluating the conjunction. If, however, the sequential conjunction begins with a user-defined relation call, or a non-compilable primitive, the process initially spawns a child process to evaluate the first literal in the conjunction. It then becomes an *AND* process. Next, it sets its continuation to the code to execute if the literal being currently evaluated, succeeds. For example, the process evaluating the conjunction:

$$A \ \& \ B$$

first spawns a process for evaluating the literal A. It then becomes an *AND* process. Finally, it sets its continuation to the code that will spawn a process for evaluating the literal B.

If a parallel conjunction of literals, i.e., "," separated literals, is to be evaluated, the process evaluating the conjunction spawns child processes to evaluate the literals in the conjunction. These processes will be evaluated in parallel. The spawning process becomes an *AND* process. It then sets its continuation to the code to execute if all the child processes succeed. For example, the process evaluating the conjunction

$$(A, B) \ \& \ C$$

spawns two processes, one for evaluating the literal A, and the other for evaluating the literal B. The continuation of the spawning process is set to the code that will spawn a process to evaluate the literal C.

If a parallel conjunction occurs at the end of a clause body, the *tail forking optimization* is applicable. This optimization is a generalization of the *tail recursion optimization* applicable to sequential logic programs. Consider the following:

$$R - P_1, P_2.$$

$$P_1 - B_1 \ \& \ B_2 \ \& \ \cdots \ \& \ (P_{11}, P_{12}).$$

After successful evaluation of the sequential conjunction $B_1 \ \& \ B_2 \ \& \ \cdots \ \& \ B_n$, the *AND* process that evaluated this conjunction can proceed by spawning two child processes—one for evaluating the literal P_{11}, and the other for evaluating the literal P_{12}. However, there is no need to increase the depth of the process tree. Since the parent of this process, i.e. the process evaluating the conjunction (P_1, P_2), is already an *AND* process, the child processes for evaluating P_{11} and P_{12} can be attached as children of that process. There would then be an AND process evaluating the conjunction (P_{11}, P_{12}, P_2).

The tail forking optimization is very important since it prevents a steady increase in the depth of the process tree during a long evaluation. Such an increase would tend to occur, for example, in the evaluation of recursive calls.

It is useful to observe that an *AND* process is created even when a conjunction consisting of just a single literal is to be evaluated. Likewise, an *OR* process is created to control the search for a candidate clause even when there is just one clause defining a relation.

We will now trace the evaluation of a PARLOG query, say $:A \cdot B$. The evaluation proceeds as though the query was $:Q$, with a user-defined relation Q, defined by $Q - A, B$. A process, say P_1, is created to evaluate the conjunction consisting of the single literal Q. Since P_1 is to evaluate a conjunction, it becomes an *AND* process. Since Q is assumed to be a user-defined relation, P_1 forks, creating a child process, say P_2, to evaluate the literal Q. P_2 becomes an *OR* process in order to search for a candidate clause. Since there is only one clause defining Q and its guard is empty, P_2 commits to that clause. That is P_2 becomes an *AND* process evaluating the conjunction of literals A and B. P_2 then forks, creating two processes to evaluate the literals A and B. These latter processes become *OR* processes in order to search for a candidate clause. After committment, they become *AND* processes evaluating the conjunction of body literals. The evaluation of the query continues in this manner until the *AND* process at the root of the *AND/OR* process tree, viz., P_1, either succeeds or fails.

Since the control structures change the state of the *AND/OR* process tree, they can be regarded as the control instructions of the abstract *AND/OR* process model. In order to support PARLOG's control structures, three additional control instructions are needed: *success, commit,* and *fail.* These are described in the following paragraphs.

Success corresponds to the successful evaluation of a single literal or a conjunction of literals. The parent of a succeeding process, i.e., one evaluating either a single literal or a conjunction of literals, always has a process type of *AND*. If the succeeding process has no siblings, it is disposed of and its parent resumes execution at its continuation. If there is no continuation, the parent process reports success to its parent. If the succeeding process has siblings, it is simply disposed of.

Commit corresponds to the ":" operator of PARLOG. We have already discussed the commit operation when we discussed the evaluation of a conjunction of guard literals.

Fail corresponds to the *FAIL* instruction of PARLOG as well as failure in the evaluation of any literal. The effect of failure depends upon the type of the parent process.

If the parent of a failing process is an *OR* process, the failing process is a process evaluating a clause guard. If the failing process has no siblings, it is disposed of, and its parent is reactivated at its continuation. If there is no continuation, the parent process reports failure to its parent. If the failing process has siblings, it is simply disposed of.

If the parent of a failing process is an *AND* process, the failing process is a process evaluating a literal in a conjunction. In this case the entire conjunction fails. The failing process and its siblings, if any, are disposed of. The parent process reports failure to its parent.

3.1.2. Data Objects and their Representation

Data objects in PARLOG are called *terms.* A term is a *constant,* a *variable,* or a *structured term.* A structured term is an n-tuple, optionally prefixed by a *functor.* The components of the n-tuple, in turn, are terms. An example of a structured term is

$$F(t_1, t_2, ..., t_n)$$

where, F is the functor. The *arity* of a structured term is the number of components in its tuple.

Notice that a structured term looks like a function call, the functor being the function name, and the tuple, the call arguments. Indeed, we will refer to structured terms as structures, and the components of the tuple as the arguments of the structure.

As another point of terminology, we will refer to the memory where terms are created during run-time as *term memory;* and the memory that contains the program as *clause memory.*

We will now describe how constants, structures, and variables are represented. Constants are represented as:

TCON	type	contents

type	Integer, real, atom, string
contents	Integer or real. Atoms and strings are also represented as integers.

Structures are represented as:

TSTR	functor − name	arity	arg − pointer

arg − pointer	Pointer to the first argument of the structure. The arguments of a structure are contiguous.
functor − name	Integer, identifying the name of the structure.
arity	Number of arguments in the structure.

Variables are represented as follows:

TVAR	b − ub − tp	l − nl	diffvar

b − ub − tp	BOUND, if the variable appears in term memory, and it is bound to a structure or another variable.
	UNBOUND, if the variable appears in term memory, and it is uninstantiated.
	TEMPLATE, if the variable appears in clause memory.
l − nl	This field would be needed only in parallel architectures.
	LOCAL, if the variable appears in term memory, is bound to another data object, and that data object is local, i.e., in the same PE; if it is uninstantiated; or if it appears in clause memory.

diffvar

NON-LOCAL, if the variable appears in term memory, is bound to another data object, and that data object is in some other PE.

If $b-ub-tp$ = BOUND, this field contains the address of the data object that the variable is bound to.

If $b-ub-tp$ = UNBOUND, this field contains a pointer to a data structure called a *demand list*, attached to the variable. When a reference to an uninstantiated variable is made, a demand for it is enqueued in the demand list. The enqueued demand contains all information necessary to restart the computation that caused the reference. Should the variable get instantiated to a non-variable term, the computation corresponding to each enqueued demand is re-scheduled for execution. Demand lists, thus, are the mechanism we use to implement suspension. The basic idea behind demand lists is exactly the same as *I*-structures [Thom80]; however, the implementation details are different.

$b-ub-tp$ = TEMPLATE implies that the variable is in clause memory. In this case, the *diffvar* field contains the variable's ID number, a number that uniquely identifies it within a clause.

In PARLOG, complex data objects are aggregated by nesting structured terms. An example of a complex data object is the term $F(A(B(x, y), C(a, b)), P(q, R(s)), x)$. As syntactic sugar, nested structures with functor name $CONS$ are written in list notation. Thus, $[a, b]$ is equivalent to $CONS(a, CONS(b, NIL))$. Also $[a \; b]$ is equivalent to $CONS(a, b)$, and the empty list, $[]$, to the constant NIL. We will assume that terms with empty functors, i.e.. just tuples, have an implicit functor name, $COMMA$, and write such terms as nested structures. For example, $((a, b), c, d)$ is equivalent to $COMMA(COMMA(COMMA(a, b), c), d)$.

In our computational model, complex (and simple) data objects are Directed Acyclic Graphs (DAGs). Figure 3.1 shows several examples. The leaves of the DAG, i.e., the nodes with outdegree zero, correspond to variables and constants. The indegree of leaf nodes may be greater than one. Such would be the case for variables that occur more than once in a term (see figure 3.1b). The nodes with outdegree greater than zero, are non-leaf nodes. Their indegree is restricted to be one. They correspond to functor names.

Terms will be assumed to be globally addressable, i.e. there is a single system wide virtual memory; Addresses will consist of two components: a PE number and a within PE address. The motivation for using a single global address space is that stream-*AND* parallelism is very difficult to implement in the absence of shared memory. This is because, if a variable v is shared among several processes, it is not known in advance which process will bind it. When v does get bound by some process, its binding will have to be communicated to all the processors where the other processes sharing v
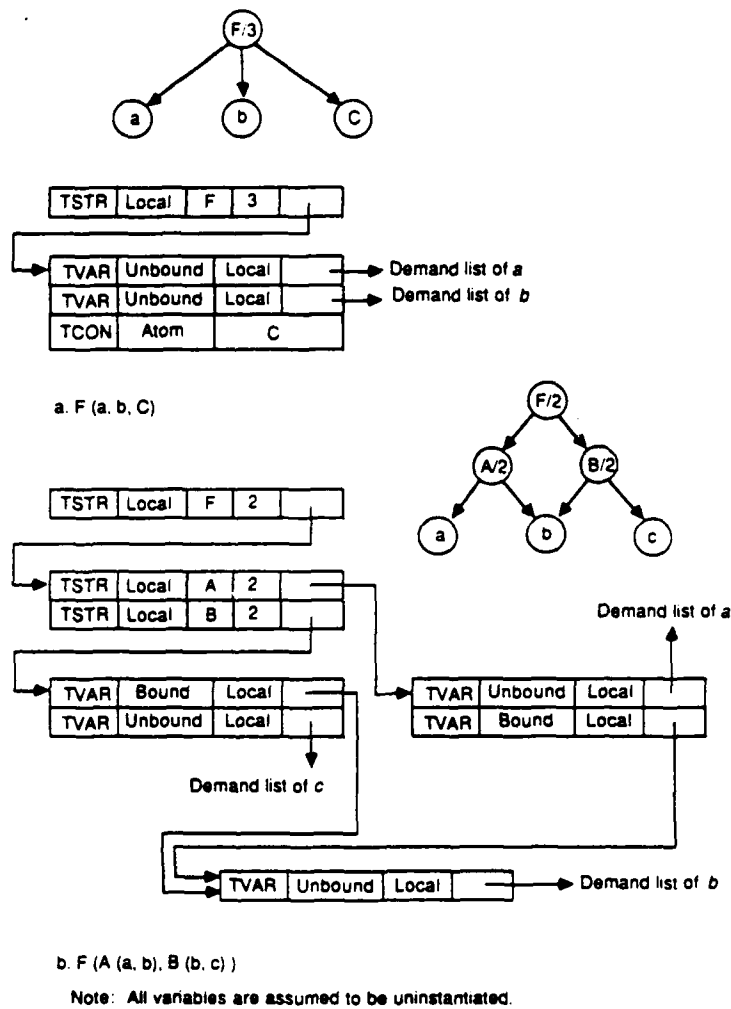
Figure 3.1. DAG Representation of Terms

reside. This communication can get very complicated in the absence of shared memory.

Thus far, we have described data objects that are part of the language definition. However, during execution of the language, data objects, called *process descriptors*, which are not part of the language definition, are created and destroyed. Process descriptors are data structures representing the abstract *AND/OR* processes. Indeed, whenever we use the term "process", we are actually talking about a process descriptor data structure. We will refer to the memory in process descriptors are created as *descriptor memory*.

We briefly explain the meaning of the different fields comprising a process descriptor. Their meaning will become more clear in section 3.2, when we describe execution of PARLOG programs on the abstract machine.

isrunning	This field serves as a lock on the process descriptor. The reason such a field is necessary is that, in the interest of parallelism, we allow growth and pruning of the process tree to proceed concurrently. The lock serves to synchronize these operations. It is set whenever a node is added to the process tree; it is reset whenever the process tree is pruned.
killflag	This field is set if the process descriptor is locked (i.e., the *isrunning* field is set), but it is to be pruned away.
state	This field indicates whether the process is an *AND* process or an *OR* process.
parent − state	Same for the parent of this process.
child − count	For an *AND* process, this field indicates the number of literals in the conjunction being evaluated by the process. This includes user defined relation calls as well as calls to system primitives. For an *OR* process, it indicates the number of clauses being searched under its control.
owu − count	For an *AND* process, this field indicates the number of needed variables appearing on the left of "$<=$" calls in the conjunction. A variable appearing on the left of a "$<=$" is said to be not needed in the clause containing the "$<=$", if it appears in no other literal in the clause. The diffvar field of such variables is set to NOT-NEEDED, instead of their ID number. For an *OR* process, this field is irrelevant.
PID	Process ID. This field is comprised of two parts: a PE number, and a process number. Thus, every process has a unique, system wide *PID*. *PID*s are not reusable.
pPID	This field contains the process ID of the parent process. The *PID* and *pPID* fields link up the *AND/OR* process tree.
continuation	For an *AND* (*OR*) process, this field indicates the code to execute when all its child processes have succeeded (failed).
body − pointer	This field has relevance for an *AND* process evaluating a guard, in which case it contains the address of the body. This information is needed in case the evaluation commits to the clause whose guard is being evaluted by the *AND* process. For an *OR* process, this field is irrelevant.
lit − num	This field indicates which child this process is of its parent.
child − list	This field is a list containing the PIDs of the children of this process.

head – size   For an *OR* process. this field contains the number of arguments in the literal (relation) being evaluated.

Another object that is not part of the language definition is the *pointer – vector*. The pointer vector is a vector of values and addresses. It serves as the binding environment for evaluation of literals and conjunctions.

For an *OR* process, the size of this vector is equal to the number of arguments in the literal (relation) being evaluated. The contents of the pointer vector are the call arguments. Literal arguments that are instantiated to constants are passed by value. Uninstantiated literal arguments and those instantiated to structures, are passed by reference. Thus, each slot in the pointer vector may contain either a constant or an address.

For an *AND* process, the size of the pointer vector is equal to the number of arguments in the head plus the number of local variables in the clause. After local variables are allocated, their addresses are inserted into the pointer vector. Should a local variable get instantiated to a constant, its pointer pointer vector slot will be replaced by that constant. Should it get instantiated to something other than a constant, its pointer vector slot is replaced by the address of the object it got instantiated to. The single assignment property of PARLOG obviates the need for consistency checking of variable bindings present in the pointer vector and those present in term memory.

Other objects that are not part of the language definition, but which are created and destroyed during execution of PARLOG programs are: *one way unification state,* and *test unification state.* We defer the description of these objects till section 3.2, where we describe how one way unification and test unification are performed in the abstract machine.

3.1.3. Program Representation

PARLOG programs are represented as data, in order to facilitate evaluation of the metacall. Literals are represented as structures, with functor name being the name of the literal, and arity equal to the number of arguments of the literal. A sequential conjunction of literals is represented as a nested structure, with an implicit functor name *AMPERSAND* and arity 2. A parallel conjunction of literals is treated similarly, except that the implicit functor name is *COMMA*. As an example, the parallel conjunction $A(x, y), B(y, z), C(x)$, is represented as $COMMA(COMMA(A(x, y), B(y. z)), C(x))$. "," and "&" are assumed to be left associative. Also, "," is assumed to bind tighter than "&".

We refer to a DAG whose root is *COMMA*, *AMPERSAND*, or a literal name, as a *literal tree*. The leaves of a literal tree are variables.

The arity of the *COMMA* functor is set to PARALLEL CONJUNCTION if the literal tree beneath it does not contain any *AMPERSAND* functors. Otherwise, it is set to SEQUENTIAL CONJUNCTION. As will be seen later, this information is used to determine whether the tail forking optimization is applicable or not.

We now discuss the representation of the PARLOG unification primitives, $"<="$ and $"="$. $"<="$ is represented as a structure with functor name $ONE-WAY-UNIF$. Its arity is set equal to the number of variables in the left argument that are needed in the clause. As explained in the previous section, a variable appearing on the left of a $"<="$ is said to be not needed in the clause containing the $"<="$, if it appears in no other literal in the clause. As will be seen later, the count stored in the arity field of the $ONE-WAY-UNIF$ functor will be used to determine whether a conjunction of literals containing $"<="$ has succeeded.

In $t_1 <= t_2$, since t_1 and t_2 can be of type $TSTR$, $TCON$, or $TVAR$, there are 9 cases possible. However, we transform the $"<="$ call so that a variable appears as the left argument of the call, or as the right argument, or as both arguments. As an example of the kind of transformation implied, $nt_1 <= nt_2$, where nt_1 and nt_2 are non-variable terms, is transformed to $nt_1 <= v$, $v <= nt_2$, v a variable.

The same transformation is done in the SPM as well. However, in the SPM, further transformations are made, which compile the one way unification to a sequence of primitive instructions. The transformations above merely change the syntactic form of calls. On the other hand, the transformations in the SPM introduce significant efficiency benefits by reducing the run-time overhead for a sequential architecture. However, sequential execution of the primitive instructions might obscure parallelism possible if v is distributed among different PEs. In order not to lose this parallelism, we choose not to compile one way unifications down to low levels.

We transform test unification calls ($"="$) so that both arguments are variables. For example, $[a, b] = v$ is transformed to $w = v$, $w <= [a, b]$; $[a, b] = [c, d]$ is transformed to $w = v$, $w <= [a, b]$, $v <= [c, d]$. $w = v$ is then represented as a structure with functor name TUM and arity 2. If one of the arguments is ground, the test unification call is equivalent to the one way unification call. For example, $[A, B] = v$ is equivalent to $[A, B] <= v$. If both arguments are ground, the test unification can be performed at compile time, flagging an error if necessary.

Clauses are represented as structures, with functor name $BACKARROW$. The arity of the $BACKARROW$ functor is set equal to the number of local variables in the clause. The first argument of this functor is the clause head, a literal. The second argument of the $BACKARROW$ functor is the clause guard, which can be either empty, or a conjunction of literals. If the guard is empty, the second argument is the constant EMPTY, otherwise it is a structure whose functor name is either $COMMA$ or $AMPERSAND$. The third argument of the $BACKARROW$ functor is the clause body, which is represented the same way as the clause guard.

Relations are sets of clauses composed with the operators "." and ";". A "." (";") separated set of clauses is represented as a nested structure with implicit functor name DOT ($SEMICOLON$) and arity 2 (see figure 3.2). "." is assumed to bind tighter than ";".

It should be clear that any DAG with root DOT or $SEMICOLON$ would have to have at least one sub-DAG rooted with $BACKARROW$. We refer to the sub-tree,

```
(R (x, y) ◄——— (A (x) & B (z) ). C (y,z)
R (x, y) ◄——— Data (x)    A (x, y) )
R (x, y) ◄——— A (x, y)  :
```
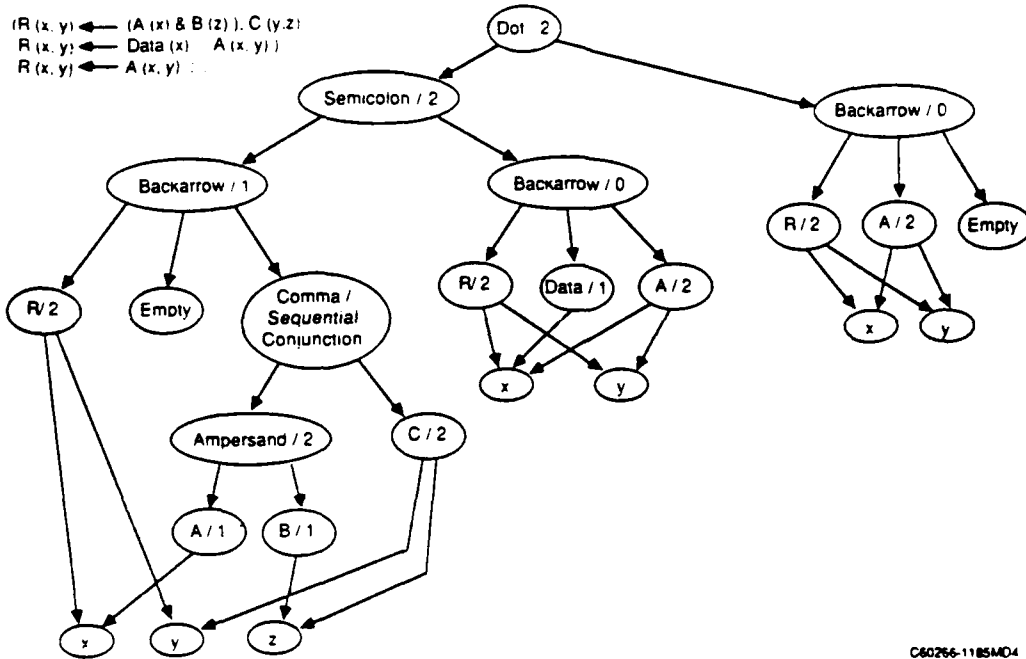
C60266-1185MO4

Figure 3.2. Example of program representation

beginning with the root and including the *BACKARROW* functors, as a *clause tree*. Thus, a *clause tree* has either a single node, the *BACKARROW* functor; or several nodes in which case its root is the *DOT* or *SEMICOLON* functor, and its leaves are *BACKARROW* functors.

The neutral composition operators, ".." and "and", are replaced by "." and "," respectively.

The arguments of literals are guaranteed to be variables or constants. This is because we replace literals like $A(B(x, y), C(D(e, f), g))$ by $A(a, b)$, $a <= B(x, y)$, $b <= C(D(e, f), g)$. The result of this transformation is that the unification related functors, $ONE - WAY - UNIF$, and TUM, are the only ones that can have structures as arguments.

The PARLOG system primitives (Less, Times, Plus, Lesseq, Call, etc.) are compiled so as to ensure that their arguments are instantiated at the time the primitive is evaluated. For example, $Call(x)$ is transformed to $DATA(x)$ & $CALL(x)$; $Times(x,y,z)$ is transformed to $(DATA(x), DATA(y))$ & $TIMES(x,y,z1)$ & $z <= z1$.

3.1.4. Operations

In this section, we motivate the operations needed to execute PARLOG programs in order to motivate them. We will give a detailed description in section 3.2, when we describe the abstract machine.

The operations needed to execute PARLOG programs fall under two categories: unification and operations that implement the PARLOG control structures. They are:

- process tree growth;

- output matching;

- input matching (i.e., one way unification),

- test unification; and

- process tree management.

Process tree growth and process tree management implement the control structures of PARLOG. Process tree growth is the creation of process descriptors. Process tree management is the execution of the control instructions, commit, success and fail, described in section 3.1.1. That is, process tree management is the pruning of *AND/OR* process tree. The reason for separating process tree growth and process tree management is to generate more parallelism.

The other three operations are related to unification. Output matching is the evaluation of a "$<=$" call in which the left argument is a variable. It is the operation by which terms in PARLOG programs are created. One way unification is the evaluation of a "$<=$" call in which the left argument is a non-variable term. Test unification is the evaluation of an "$=$" call.

3.2. A Parallel Abstract Machine for PARLOG

In this section, we describe a parallel abstract machine for executing PARLOG programs. We call this abstract machine $(AMP)^2$—Asynchronous Message-passing based Parallel Abstract Machine for PARLOG.

The architecture of $(AMP)^2$ is shown in figure 3.3. $(AMP)^2$ consists of abstract processing elements (PEs) linked together by an abstract network, called the *inter-PE interconnection network*. Each abstract PE is a collection of the following computing agents: User Interface (*UI*), Network Send (*NS*), Network Receive (*NR*), Process Tree Manager (*PTM*), Process Tree Grower (*PTG*), Input Matcher (*IM*), Test Unifier (*TU*), Output Matcher (*OM*), Data Checker (*DC*), Var Checker (*VC*), and Term Memory Allocator (*TMA*). Each agent performs a dedicated function. The *PTM, PTG, TU, IM* and *OM* agents respectively perform the five operations we motivated in section 3.1.4: process tree management, process tree growth, test unification, input matching, and output matching. The *DC* and *VC* agents respectively evaluate the *DATA* and *VAR* system primitives of PARLOG. The *DATA* primitive succeeds if its argument is instantiated to a non-variable term. Otherwise, the call suspends. It can never fail. The *VAR* primitive checks whether its argument is instantiated to a variable at the time of the call. If so, it succeeds, else it fails. The *NS* agent receives messages from other agents in the PE, and sends them out to the network. The *NR* agent receives messages from the network and routes them to the appropriate agents in the PE. The *UI* is present only in PE_0 and is the agent through which communication with the user (either humans or application programs) is done.
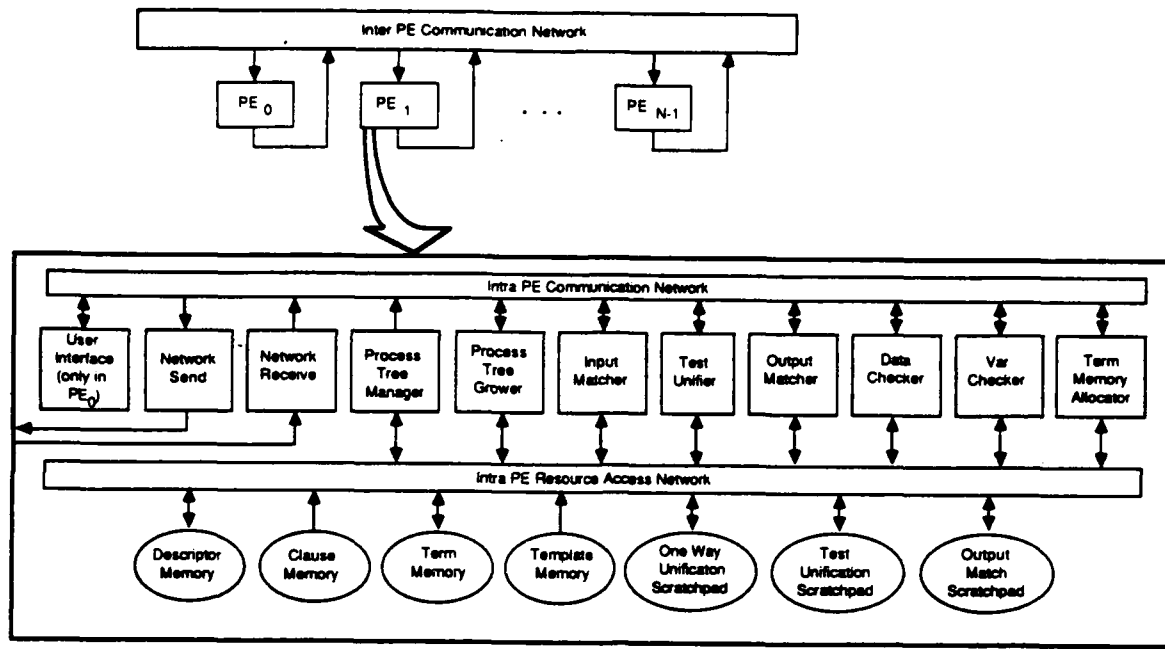
```
C50266-1186MD2
```

Figure 3.3. (AMP)^2 architecture

An agent can communicate with other agents by sending messages over the abstract *intra-PE communication network*.

It can also access one or more of the following resources: Descriptor Memory, Clause Memory, Term Memory, Template Memory, One Way Unification Scratchpad, Test Unification Scratchpad, and Output Match Scratchpad. Agents access resources via the abstract *intra-PE resource access network*.

Each agent has a mailbox, in which it receives messages. It processes messages in its mailbox, independently of the other agents. Thus, parallelism is exploited at two levels in (AMP)^2—several PEs processing concurrently, and several agents within a PE processing messages concurrently.

(AMP)^2 is a loosely coupled multiprocessor. However, it implements a shared memory abstraction. That is, PARLOG programs executing on (AMP)^2 see a single address space. Why is providing a shared memory abstraction on a loosely coupled architecture important? There are two opposing factors that come into play in PAR-LOG implementations. On the one hand, a loosely coupled architecture is attractive for performance reasons. On the other hand, a shared memory architecture is attractive for stream-*AND* parallelism, since it is very difficult to implement stream-*AND* parallelism on loosely coupled architectures [Greg85]. The major feature of (AMP)^2 is that it reconciles these opposing factors by providing a very efficient shared memory abstraction on a loosely coupled architecture—efficient in the sense that the memory contention and memory latency problems associated with shared memory systems are absent.

$(AMP)^2$ is an abstract architecture and not a physical hardware architecture. Its purpose is to lay bare all the functions that need to be performed in order to execute PARLOG programs. It thus acts as a functional specification for the hardware architecture. $(AMP)^2$ features only logical entities. For example, the abstract networks are logical communication channels. These channels would have to be implemented via appropriate interconnection networks (e.g., bus, banyan, shuffle, hypercube, etc.) in a physical hardware architecture. Likewise, the agents are also logical entities. They would have to be mapped onto physical hardware in order to get a physical hardware architecture. There are several possibilities for this mapping. In a one-one mapping, there is a dedicated hardware unit for each agent. In a many-one mapping, one hardware unit performs functions represented by several agents. Finally, in a one-many mapping, several identical hardware units are dedicated to a single function.

3.2.1. Executing PARLOG programs on $(AMP)^2$

The user's query is read in by the User Interface (UI) agent, which is present only in PE_0. A query of the form $:A(x, y), B(y, z)$ is treated as though there were a user defined relation called $QUERY$ defined by the clause $QUERY - A(x, y), B(y, z)$. The UI agent creates an AND process, the root of the AND/OR process tree. *In the following discussion, creation or deletion of a process means creation or deletion of the process descriptor.* It does not mean creation or deletion of a process in the operating systems sense. Next, the UI agent sends out a $SPAWN$ message to the Network Send (NS) agent. A $SPAWN$ message is created whenever a user defined literal is to be evaluated. It includes the PID of the AND process evaluating the conjunction that the relation call is part of, and a pointer vector containing the call arguments. The NS agent sends the $SPAWN$ message to the inter-PE communication network. In the current version of $(AMP)^2$, the network uses a uniformly distributed random number generator to route the $SPAWN$ message to some random PE. In later versions, the network may incorporate sophisticated load balancing strategies.

Process Tree Growth

The $SPAWN$ message is routed by the Network Receive (NR) agent of the destination PE to the Process Tree Grower (PTG), which creates an OR process to search for a candidate clause. Send/acknowledge protocols are used to record parent child relationships, which may exist across several PEs.

The PTG traverses the DAG of the relation being evaluated. Recall that each PE has a copy of the program in its clause memory. On encountering a $SEMICOLON$ functor during this traversal, the PTG creates an OR process descriptor in the descriptor memory. The PTG sets the *continuation* field of this process to the address of the right subtree of the functor, and then recursively traverses the left subtree. For an OR process, the continuation indicates which clause(s) to search next, if the guard of the current clause fails. The continuation field in an OR process descriptor implements the ";" control construct of PARLOG. On encountering a DOT functor during the traversal, the PTG simply recurses on the left and right subtrees.

Eventually, the *PTG* encounters a *BACKARROW* functor. At this point, it is ready to evaluate the guard of a clause. If the guard is a conjunction of more than one literal (i.e., the third argument of *BACKARROW* is either *AMPERSAND* or *COMMA*), it creates an *AND* process to evaluate the guard. However, if the guard contains just a single literal, it does not create an *AND* process for it. This is an important optimization. It prevents the process tree from growing unnecessarily. The SPM also features this optimization. In addition, in the SPM, the process tree does not grow if the guard doesn't suspend (on one way unification), even if it is a conjunction of more than one literal. The sequential nature of the SPM makes this latter optimization possible: a conjunction of one way unification calls is actually evaluated one at a time, and so, if none of the calls suspend, there is no need to create an *AND* process. On the other hand, in a parallel architecture, it is desirable to evaluate a conjunction of one way unification calls concurrently. It is for this reason, the *PTG* does not perform the latter optimization: it creates an *AND* process even if the guard contains only a conjunction of one way unification calls, none of which may suspend.

In a parallel machine, the *AND* processes corresponding to guards of different clauses in a relation, may be created on different PEs. This is how committed *OR* parallelism is exploited.

Prior to evaluating the guard of a clause, the *PTG* requests the Term Memory Allocator (*TMA*) to allocate space for the local variables in the clause. It uses the addresses returned by the *TMA* to create a pointer vector, i.e., the binding environment for the evaluation. It then continues traversal of the relation's DAG. Traversal of the DAG within a clause is similar to the traversal of the DAG of a relation, except that *AND* processes are created, and *AMPERSAND*s and *COMMA*s are encountered instead of *SEMICOLON*s and *DOT*s respectively.

Eventually, the *PTG* encounters a user defined literal, a system primitive, or a unification related call. To evaluate user defined literals, the *PTG* sends out a *SPAWN* message, which is processed at some random PE as described above. Thus, user defined literals may be evaluated in different PEs, concurrently. System primitives do not cause a *SPAWN* message to be sent. The DATA and VAR system primitives are evaluated by the Data Checker (*DC*) and Var Checker (*VC*) agents respectively. Their implementation is quite straightforward, and we will not describe it here. The other primitives are evaluated directly by the *AND* process that controls the conjunction the primitives are part of.

In our description of the unification operations in $(AMP)^2$, v, w, x, y, and z denote variables, nt denotes a non-variable term, and t denotes either a variable or a non-variable term.

Output Matching

$v <=t$, called output matching, is the means by which PARLOG terms are created. When the *PTG* encounters such a call during its traversal, it sends a message to the PE in whose address space v is resident to evaluate the call. The message includes the call arguments. The call arguments depend upon the compile time form of

t. If the compile time form of t is a variable, the call arguments are the term memory addresses of v and t. Otherwise, the call arguments are the term memory address of v and the clause memory address of t. The message is routed to the Output Matcher (OM) agent by the NR of the destination PE. If v is not an uninstantiated variable at the time of the call, the OM flags a run-time error. If, at compile time, t is a variable, the OM binds v to t. Otherwise, it creates an instance of t and binds v to that object. After binding v, the OM restarts the computations enqueued on v's demand list, by sending appropriate messages.

In general, t in $v <= t$ may be in some other PE's address space. For example, t may be a local variable in some other clause evaluated in the other PE; or it might have been created via a $<=$ call in that clause. Thus, structured terms in $(AMP)^2$ may be partitioned across several different PEs.

One Way Unification

The distribution of terms across several PEs, coupled with the fact that each PE has a copy of the program, makes possible a parallel algorithm for one way unification. When the PTG encounters a one way unification call, say $nt <= v$, it sends a message to the PE in whose address space v is resident to evaluate the call. The message identifies the AND process evaluating the conjunction that the call is part of. Let PID denote its process descriptor. The message is routed to the Input Matcher (IM) by the NR of the destination PE. We refer to the left argument of the one way unification call, as the *template DAG* and the right argument as the *term DAG*. We refer to the partitions of the term DAG as *term sub − DAGS*. Templates, the left arguments of all one way unification calls, are loaded at compile time into the template memory of each PE.

The IM traverses, in lock step, the term sub-DAG and the template DAG. Its behavior on encountering a constant or a structured term in the template DAG is similar. We describe its behavior for the former case. There are three possibilities:

i). The term sub-DAG is a non-variable term or is a variable bound to a non-variable term. In this case, the IM checks if the non-variable term is a constant equal to the one in the template DAG. If so, the traversal succeeds; otherwise it fails and the IM sends a *FAIL* message to the PE denoted by PID.

ii). The term sub-DAG is a variable, but it is bound to an object in some other PE. An *access fault* is said to occur in this situation. On encountering an access fault, the IM sends a message containing all necessary information to the other PE asking it to continue the one way unification. It doesn't issue a memory request over the network to fetch remote data, and wait for it to arrive. This is a very important optimization: rather than waiting for remote memory requests to be resolved, the IM sends the computation to where the remote data is, and then starts processing the next message in its mailbox.

Actually, the IM does not send the continuation message as soon as it detects an access fault. Instead, it posts an entry in a data structure called the *one way*

unification state, containing the following information: the template DAG and term sub-DAG node addresses at which the one way unification is to continue in the other PE, *PID*, and the PE where the access fault occurred. This data structure is stored in the one way unification scratchpad. The reason for posting such an entry rather than sending the computation right away is that the traversal might fail in this PE, in which case there is no need to continue the computation. This is another very important optimization. Because of this optimization, a process will suspend on remote variable accesses, only if the $<=$ call cannot succeed or fail otherwise. Later, if the traversal succeeds, the *IM* will send the computation out. There is a speed penalty that we have to pay in this approach, since the one way unification cannot continue in other PEs until the traversal of the entire sub-DAG present in this PE has been completed. However, the speed penalty is likely to be small, since the sub-DAGs are more than likely to be small. Besides, the speed penalty has to be weighed against the cost of communicating over the network.

During the course of the traversal, several entries may be posted in the one way unification state. The more PEs where the unification is to continue, the more the parallelism.

The one way unification can be considered to have succeeded only if the traversal of every sub-DAG is successful. The successful traversal of a sub-DAG is called *partial success*. Messages denoting partial success are propagated in the direction opposite to which the continuation messages are propagated. This is why continuation messages identify the PE where the access fault occurred.

iii). The term sub-DAG is an uninstantiated variable. In this case, the *IM* posts a demand for the value of this variable in the one way unification state. It doesn't enqueue the demand right away on the variable's demand list; later, if the traversal succeeds, it will enqueue the demand. Because of this optimization, a process will suspend on a variable only if the $<=$ call cannot succeed or fail otherwise. Enqueueing demands is how suspension in one way unification is implemented in $(AMP)^2$.

If a variable in the template DAG is encountered, the *IM* posts an entry in the one way unification state indicating that a binding for a variable in *nt* has been found. If the traversal succeeds, the *IM* sends out the binding to the PE denoted by *PID*. The variable would have been allocated in the term memory of that PE. When the binding reaches that PE, the *OM* agent of that PE binds the variable.

Test Unification

When the *PTG* encounters a test unification call, $t_1 = t_2$, it sends a message to the PE in whose address space t_1 resides. The Test Unifier (*TU*) agent in that PE processes this message. The evaluation of test unification calls is similar to the evaluation of one way unification calls. The only difference is that no variables are bound. Also, there is no notion of a template DAG since the arguments of the call are not known at compile

time. Therefore, access faults can occur on both sub-DAGs during traversal. If an access fault occurs when traversing the right sub-DAG, the TU posts a demand for the right sub-DAG node in a data structure called the *test unification state,* which is stored in the test unification scratchpad. The demand is basically a remote memory request. If the traversal succeeds, the TU sends the remote memory request out on the network. It also sends the context necessary to restart the computation when the remote data arrives. This is a very important optimization: by sending context along with remote memory requests, the TU does not have to wait for remote data to arrive; rather, it starts processing the next message in its mailbox. If an access fault occurs when traversing the left sub-DAG, the TU posts an entry indicating that the evaluation of the call is to be continued in some other PE. Later, if the traversal succeeds, the TU sends continuation messages out.

Process Tree Management

During the course of the evaluation of a PARLOG query, the AND/OR process tree grows and shrinks as literal evaluations (and conjunction evaluations, and clause searches) succeed and fail. The PTG grows the AND/OR process tree; the PTM prunes it. The PTM and the PTG can asynchronously access the descriptor memory, thus increasing parallelism. However, concurrent access to a process descriptor must be synchronized. Therefore, each process descriptor has a lock field.

Process tree management is essentially the processing of $SUCCESS$ and $FAIL$ messages. A $SUCCESS$ ($FAIL$) message is sent by a process to its parent if it succeeds (fails). Let $CPID$ denote the succeeding process, PID its parent, and $PPID$ the parent of the parent. The behavior of the PTM upon receipt of a $SUCCESS$ message depends upon the type of the PID process and the $PPID$ process

i). PID and $PPID$ are OR processes. This occurs when a group of nested clauses separated by ";" and "." is being searched for a candiate clause. The PTM destroys PID and recursively, all of PID's children. It then sends a $SUCCESS$ message to $PPID$.

ii). PID is an OR process, $PPID$ is an AND process. This occurs when $CPID$ is a guard that has succeeded, signifying that a candidate clause has been found. The PTM sends a $SPAWN-AND$ message, which contains $PPID$ and a pointer to the body of the candidate clause, out on the network. It then destroys PID and recursively, all of PID's children, which are the siblings of $CPID$. There may be $SUCCESS$ or $FAIL$ messages in transit from the sibling guards. However, these messages will be ignored when they arrive at their destination, since PID would have already been destroyed. To guarantee that the destination process is destroyed, it is necessary to ensure that PIDs are not reused. This presents no problems: all that is needed is that each PE use unique process numbers for its PIDs. It can be seen that the PARLOG commit operator, ":", is implemented in $(AMP)^2$ by executing the body of the *first* clause whose guard terminates successfully. The graph reduction should also be apparent: the process evaluating a literal is reduced to another process evaluating the body of the candidate clause.

The *SPAWN—AND* message is routed by the network to a random PE, just as the *SPAWN* message. The *SPAWN—AND* message is processed by the *PTG*. The *PTG* first checks if the *tail forking optimization* is applicable. If not, the *PTG* creates an *AND* process and sends a *SPAWN—AND—ACK* message to the PE containing *PPID*. Otherwise, the conjunction of body literals is controlled by *PPID* itself—without creating a new *AND* process. This optimization is a generalization of the tail recursion optimization in traditional logic languages [Greg85]. In a sequential implementation of PARLOG, the tail forking optimization applies if there is a parallel conjunction at the end of a clause. However, (AMP)^2 being a parallel machine, we insist on an additional condition: *PPID* should be in the same PE to which the network routed the *SPAWN—AND* message. In the absence of the second condition, we might get a single *AND* process with lots of child processes. While this is fine in a sequential machine, it leads to an unacceptable bottleneck in a parallel machine.

iii). *PID* is an *AND* process. If *PID* does not have a parent, i.e., if *PID* is the root of the *AND/OR* process tree, the query evaluation is complete. Otherwise, the *PTM* destroys *PID* and recursively. all of *PID*'s children. It then sends a *SUCCESS* message to *PPID*.

Failure handling is quite straightforward. If *PID* is an *AND* process, the *PTM* destroys *PID* and recursively, all of *PID*'s children. It then sends a *FAIL* message to *PPID*. If *PID* is an *OR* process, the *PTM* checks if *PID* has a continuation. If so, it sends a message to the *PTG* to initiate the evaluation of the guard of another clause. If *PID* does not have a continuation, the *PTM* destroys *PID* and recursively, all of *PID*'s children, and then sends a *FAIL* message to *PPID*.

3.3. Mapping (AMP)^2 onto the Connection Machine

In this section, we briefly discuss the feasibility of mapping (AMP)^2 onto the Connection Machine architecture, i.e., of executing PARLOG on the Connection Machine. As we mentioned at the beginning of this chapter, this mapping is best done after studying the run-time execution behavior of PARLOG programs, to answer questions such as: How much parallelism is there? What is the granularity of parallelism? What are the communication patterns between the agents? What operations are performed most frequently? Do they require hardware support? If so, what kind? What architectural features are required for the efficient execution of single-solution PARLOG programs? Should the PE design be simple or should it be complex? The above study is a major effort in itself and beyond the scope of this project. What we present in this section is a qualitative discussion of the feasibility of executing PARLOG programs on the Connection Machine.

The Connection Machine (CM) [Hill85], is a fine grained, highly parallel, SIMD (Sing Instruction Multiple Data) computer. It consists of 64K processors, each with 4K bits of memory and a 1 bit wide ALU. Adding two 16 bit numbers takes 16 machine cycles. The total memory capacity of the CM is 32 Mbytes.

The CM requires a front-end host computer to issue instructions. These instructions are broadcast to all the 64K processors, each of which executes the same instruction, operating on the contents of its own memory. This concept is called *data level parallelism*.

There are two forms of communication in the CM: a single bit wide global-or network and a 16 dimensional hypercube. The global-or network allows aggregate operations such as global-minimum, global-maximum, global-or, etc., to be performed quickly, while the hypercube network allows processors to communicate by exchanging packets of information.

The form of parallelism in the CM, viz., data level parallelism, is different from the form of parallelism found in control level parallel processing machines. In the latter, the programmer is required to divide his program into fragments, one for each processor. Data level parallel processing works best on problems with large amounts of data, whereas control level parallel processing works best when the ratio of program to data is high.

The CM is best suited to applications that have a large amount of data level parallelism. Examples of such applications include document retrieval from a large bibliographic database, image processing, VLSI circuit design, and fluid flow problems.

On first glance, it would appear that PARLOG has plenty of data level parallelism — different parts of a large structured term may be constructed in parallel by different processes. For example, the elements of the list $[e_1, e_2, ..., e_n]$ may be constructed in parallel by n concurrent processes. However, a PARLOG process constructs only the top level structure of a variable's binding. That is, it either binds a variable to a ground term, or if it binds it to a partially instantiated term, then the further instantiation of this term is done by *another* PARLOG process. This is a characteristic feature of stream-*AND* parallelism. A process partially instantiates a variable and passes this binding to another process through a shared variable. The second process, in turn, partially instantiates the variables in this binding, and passes them to a third process, and so on. Thus, the form of parallelism intrinsic to PARLOG is one where several different processes act on different portions of a large data structure. In other words, the form of parallelism intrinsic to PARLOG is control level parallelism with multiple threads of control.

This form of parallelism is directly opposite to what is best supported by the CM — a single process operating on all portions of a large structure concurrently. Therefore, we conclude that the Connection Machine, with its SIMD style data level parallelism, is not a good choice for executing PARLOG. Indeed our experience in designing $(AMP)^2$ supports this conclusion. A shared memory abstraction on a coarse grained, loosely coupled architecture is better suited to implementing PARLOG than the CM.

3.4. Conclusions

This chapter presented our design of a parallel abstract machine for executing PARLOG. The principal conclusion from this work is that shared memory greatly facilitates implementing PARLOG's stream-*AND* parallelism and that the key to high performance stream-*AND* parallelism is an efficient shared memory abstraction on a loosely coupled architecture. The abstract machine described in this chapter achieves this via a number of optimizations. These optimizations address critical problems in the design of efficient parallel architectures. They address the principal sources of overhead, viz., communication and memory latencies, and synchronization overheads.

These problems are overcome because agents never suspend, waiting for remote data to arrive. If a computation requires access to remote data, the agent sends the computation over to where the data is, whenever possible (as in one way unification, and sometimes, in test unification). It then processes the next message in its mailbox. If a remote memory request cannot be avoided (as sometimes happens during evaluation of test unification calls), the agent sends a context with the remote memory request, and then processes the next message in its mailbox. The context enables restarting the computation when the remote data arrives. Thus, arbitrary latencies in memory requests can be tolerated in $(AMP)^2$, which is very important for parallel architectures [Iann83].

Finally, we investigated the feasibility of executing PARLOG programs on the Connection Machine architecture. The conclusion from this work is that a coarse grained, loosely coupled architecture is better suited than the CM, since the form of parallelism best supported by the CM is directly opposite to that found in PARLOG.

CHAPTER 4

Data/Knowledge Base Query Processing Concepts

This chapter describes our investigation of the concepts relating to data/knowledge base (D/KB) query processing. It is organized as follows. Section 4.1 gives several definitions relating the composition of a data/knowledge base. Recursive query processing is a key concept that differentiates D/KB query processing from traditional database query processing. Section 4.2 introduces definitions relating to recursion. Section 4.3 describes rule representation. Section 4.4 describes top-down and bottom-up evaluation of rule-based queries. Section 4.5 describes evaluation of non-recursive rule-based queries. Section 4.6 describes the evaluation of recursive rule-based queries. Section 4.7 describes the concepts pertaining to recursive rule-based query optimization.

4.1. Data/Knowledge Base

We give several definitions pertaining to the data/knowledge base. These definitions appear in [Banc86].

The data/knowledge base is a set of Horn clauses and schemas. A Horn clause has the form

$$head \leftarrow body$$

where *head* is zero or one atomic formulas (predicates with arguments supplied) and *body* is a conjunction of zero or more atomic formulas. All arguments that are variables are implicitly unversally quantified. The logical interpretation of a Horn clause is that the body implies the head.

Example: $a(X, Y) \leftarrow b(X, Y), c(Z, Y)$ means that for all X, Y, and Z, $b(X, Y)$ and $c(Z, Y)$ implies $a(X, Y)$. []

A *relation definition* is the set of clauses whose head refers to a given relation.

A Horn clause with an empty body and no variables in its head is called a *fact*. Facts can be written with no implication sign, e.g., *parent*(a, b) means that a is the parent of b.

A *rule* is a Horn clause that is not a fact.

Predicates corresponding to facts are called *base predicates*. Predicates corresponding to the head of a rule are called *derived predicates*.

We can assume without loss of generality that a relation is defined entirely by rules or entirely by facts. If a set of clauses does not meet this condition, it can easily be transformed into a set of clauses that does.

Example: The set of clauses

$$p(X, Y) \leftarrow a(X, Z), b(Z, Y).$$

$$p(a, b).$$

$$p(c, d).$$

is equivalent to

$$p(X, Y) \leftarrow a(X, Z), b(Z, Y).$$

$$p(X, Y) \leftarrow p_1(X, Y).$$

$$p_1(a, b).$$

$$p_1(c, d). \quad []$$

Thus, the data/knowledge base can be partitioned into rule relations and fact relations. The set of rule relations is called the *intensional knowledge base*, or *rulebase*, while the set of fact relations is called the *extensional knowledge base*, or *database*. The intensional knowledge base contains only derived predicates, while the extensional knowledge base only base predicates.

The motivation for distinguishing between the intensional and extensional knowledge bases is that rules are stored in compiled form to allow for more efficient access, while facts are stored directly. We will describe different storage structures for storing the compiled form of rules later in this chapter.

The schema of a base predicate is the same as a relational database schema; it contains the names of the arguments and their types. The schema of a derived predicate is derived using the base predicate schema and the rules.

4.2. Recursion

In this section we give several definitions pertaining to evaluating recursive queries. Again, these definitions appear in [Banc86].

We will use the set of Horn clauses shown in figure 4.1 to illustrate the definitions. The b_i's in this example are base predicates, while p, q, p_1, and p_2 are derived predicates.

A derived predicate q is *reachable* from a derived predicate p if

(i) q is in the body of a rule having p as its head, or

(ii) q is in the body of a rule having s as its head and s is reachable from p.

Example: In figure 4.1 p_1 is reachable from p. So is b_1 because b_1 is reachable from p_1 and p_1 is reachable from p. []

Two derived predicates p and q are *mutually recursive* if they are reachable from each other.

Example: In figure 4.1 p and q are mutually recursive. []

A predicate p is *recursive* if it is reachable from itself.

R_1: $p(X, Y) - p_1(X, Z), q(Z, Y)$.

R_2: $p(X, Y) - b_3(X, Y)$.

R_3: $p_1(X, Y) - b_1(X, Z), p_1(Z, Y)$.

R_4: $p_1(X, Y) - b_4(X, Y)$.

R_5: $p_2(X, Y) - b_2(X, Z), p_2(Z, Y)$.

R_6: $p_2(X, Y) - b_5(X, Y)$.

R_7: $q(X, Y) - p(X, Z), p_2(Z, Y)$.

Figure 4.1: Sample data/knowledge base

Example: In figure 4.1 p_1 is a recursive predicate. []

A rule $p - p_1, p_2, ..., p_n$ is called a *recursive rule* is there exists a p_i in the body that is mutually recursive to p.

Example: In figure 4.1 R_3 is a recursive rule. []

Two rules are mutually recursive if the predicates in their heads are mutually recursive.

Example: In figure 4.1 R_1 and R_7 are recursive rules. []

A recursive rule is called a *linear recursive rule* if there is only one predicate in the body that is mutually recursive to the head. A recursive rule that is not linear is called a *nonlinear recursive rule*.

Example: All the recursive rules in figure 4.1 are linear. However, a rule of the form $p - p_1, p_1, r$ is nonlinear. []

It can be easily shown that mutual recursion is an equivalence relation on the set of derived predicates and the set of rules. Mutual recursion partitions the set of derived into disjoint blocks of mutually recursive predicates.

Example: $\{p, q\}$, $\{p_1\}$, and $\{p_2\}$ are the disjoint blocks of mutually recursive predicates in figure 4.1. []

The predicates in a block must be evaluated as a whole. Mutual recursion groups together rules needed to evaluate the predicates in a block.

Example: The rule partitions for figure 4.1 are $\{R_1, R_2, R_7\}$, $\{R_3, R_4\}$, and $\{R_5, R_6\}$. []

4.3. Rule Representation

Rules are typically represented in a graph formalism. Predicate Connection Graphs (PCGs) are an example of such a formalism. PCGs were proposed by McKay and Shapiro [McKa81] as a representation to facilitate reasoning with recursive rules. A PCG is a directed graph that represents the relationships between head predicates and body predicates. Each node in the PCG represents a predicate. Edges arise from rules. If there is a rule of the form, $p \leftarrow p_1, p_2, ..., p_n$, there is a directed edge from p to each of the p_i's. Figure 4.2 shows the PCG for the data/knowledge base of figure 4.1.
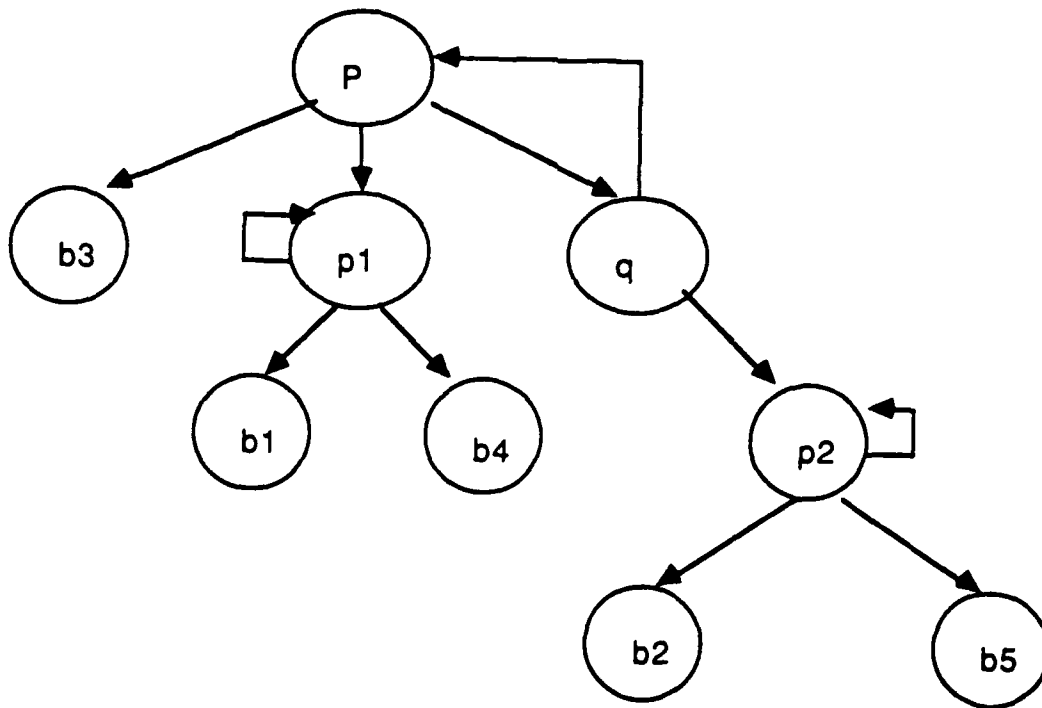
Figure 4.2. Predicate Connection Graph for figure 4.1

The definitions of the previous section can be recast in terms of the PCG. The blocks of mutually recursive predicates are the *strongly connected components* of the PCG. A strongly connected component, also called a *clique*, of a graph is a set of nodes such that there is a directed path between each pair of nodes.

In the context of D/KB query processing, we will use a somewhat broader definition of a clique. Here, by clique we will mean a set of mutually recursive predicates as well as the rules needed to evaluate these predicates. Obviously, some of these rules will be recursive. The rest are called *exit rules*. Exit rules ensure that the evaluation of a recursive predicate terminates. Therefore, every clique must have at least one exit rule.

Example: In figure 4.1 we need rules R_3 and R_4 to evaluate p_1. R_3 is a recursive rule, while R_4 is an exit rule. []

Figure 4.3 shows the cliques for the sample D/KB shown in figure 4.1.

	Recursive predicates	p, q
Clique 1	Recursive rules	R_1, R_7
	Exit rules	R_2
	Recursive predicates	p_1
Clique 2	Recursive rules	R_3
	Exit rules	R_4
	Recursive predicates	p_2
Clique 3	Recursive rules	R_5
	Exit rules	R_6

Figure 4.3: Cliques for figure 4.1

4.4. Top-Down vs. Bottom-Up Evaluation

There are essentially two strategies for evaluating Horn clause queries — top-down and bottom-up [Banc86]. We will illustrate these strategies via an example. Consider the following rules

R_1: $r(X, Y) \leftarrow s(X, Z), b_1(Z, Y)$.

R_2: $r(X, Y) \leftarrow b_2(X, Y)$.

R_3: $s(X, Y) \leftarrow b_3(X, Y)$.

R_4: $query(X) \leftarrow r(X, "a")$.

Here, the b_i's are base predicates and r, s, and *query* derived predicates. Suppose we want to evaluate the predicate *query*. In bottom-up evaluation, we start with the base predicates in the body of rules and keep combining them with other predicates in the body to produce the head predicates. We stop when *query* is generated. For the above

rules, we get s from b_3. We combine s with b_1 to get a partial result for r. We get another partial result from b_2. Finally, we take the union of these partial results to get all the values of r. We then apply a selection on r to get *query*, i.e., the values of X.

In top-down evaluation, we start with the predicate to be evaluated and keep evaluating predicates in the body of rules defining this predicate. During this process, we propagate information till we reach the base predicates. For the above rules, evaluating *query* means evaluating r with its second argument bound to "a." We use rule R_1 to propagate this binding to b_1 and R_2 to propagate it to b_2. From b_2, we get a partial result for X. From b_1, we get values for Z, which produces bindings for the second argument of s. These bindings are propagated using rule R_3 to b_3, which produces another partial result. The union of these partial results then gives all the values of X.

Bottom-up strategies are simpler and easy to implement, but they compute a lot of useless results, since they do not use knowledge about the query to restrict the search space. On the other hand, top-down strategies are more efficient since they use knowledge about the query to propagate information, but they are more complex and harder to implement.

Several optimization strategies have been proposed for use with bottom-up strategies [Beer86, Banc86 ...., Banc86, Sacc86, Sacc86 ....]. These strategies utilize query information to restrict the search space, while at the same time enjoying the ease of implementation advantage of bottom-up strategies. Therefore, we will focus only on bottom-up strategies.

4.5. Evaluating Nonrecursive Predicates

Bottom-up evaluation of a nonrecursive predicate can be achieved by a straightforward compilation to relational algebra. For example, evaluating the predicate r defined by the rules

$$r(X, Y) - s(X, Z), b(Z, Y).$$

$$r(X, Y) - c(X, Y).$$

is equivalent to evaluating the following relational algebra expression

$$c \bigcup \pi_{r.1, b.2} (s \underset{r.2=b.1}{\bowtie} b)$$

4.6. Evaluating Recursive and Mutually Recursive Predicates

Bottom-up evaluation of a recursive predicate involves computing the least fixed point (LFP) of a recursive equation [Emde76]. For example, evaluating the recursive predicate p_1 in figure 4.1, which is defined by the Horn clauses

$$p_1(X, Y) - b_1(X, Z), p_1(Z, Y).$$

$$p_1(X, Y) - b_4(X, Y).$$

corresponds to evaluating the LFP of the following recursive equation

$$p_1 = b_4 \bigcup \pi_{b_1.1, p_1.2} (b_1 \underset{b_1.2 = p_1.1}{\bowtie} p_1)$$

$$= b_4 \bigcup b_1 \bigcirc p_1$$

$$= f(p_1)$$

where the operator '$\bigcirc$' is called the *composition* operator, which is a join followed by a projection on the non-join columns.

As defined by Ullman [Aho79], a least fixed point of the equation $p_1 = f(p_1)$ is a relation $p_1{}^*$ that satisfies

(i) $p_1{}^* = f(p_1{}^*)$ and

(ii) if p_1 is a relation such that $p_1 = f(p_1)$, then $p_1{}^* \subseteq p_1$.

In general, a recursive equation may not have a least fixed point. However, if the function f is monotone in the sense that

$$if p_1 \subseteq p_2 \text{ then } f(p_1) \subseteq f(p_2)$$

it is guaranteed to have a least fixed point [Tars55].

The nice property about Horn clauses is that the function f consists only of union and composition operators and is therefore monotone. This means that recursive rules are guaranteed to have a least fixed point[1].

Algorithm 1 (see [Aho79, Banc86]) is a natural way to compute the LFP of the equation $r = f(r)$.

$$r^0 = \phi;$$

repeat

$$r^{j+1} = f(r^j)$$

until $(r^{j+1} - r^j = \phi);$

Algorithm 1: Naive Evaluation of a Recursive Equation

This method is called *naive evaluation* [Banc86]. Naive because the entire relation r^j is used to compute r^{j+1} even though the monotonocity of f and r^0 being equal to ϕ ensure that $r^j \subseteq r^{j+1}$, i.e., tuples of r^j are also present in r^{j+1}.

---

[1] Of course, it is necessary that there be an exit rule in order for the recursion to terminate.

A more efficient procedure is to compute only the difference between r^{j+1} and r^j during each iteration. This procedure is called *semi-naive evaluation* [Banc85] (see algorithm 2 below).

$$r^0 = \phi;$$

$$\delta r^0 = f(\phi);$$

while $(\delta r^j \neq \phi)$ **do** {

$$r^{j+1} = r^j \cup \delta r^j;$$

$$\delta r^{j+1} = f(r^{j+1}) - f(r^j);$$

}

Algorithm 2: Semi-naive Evaluation of a Recursive Equation

As we mentioned before, a set of mutually recursive predicates must be solved together as a whole. This involves finding the LFP of a set of recursive equations.

Example: Evaluating p and q in figure 4.1 involves finding the LFP of the following recursive equations

$$p = b_3 \cup p_1 \bigcirc q = f_1(p, q)$$
$$q = p \bigcirc p_2 = f_2(p, q) \quad []$$

In general, evaluating a set of mutually recursive predicates $r_1, ..., r_n$ will involve finding the LFP of a set of recursive equations of the form

$$r_1 = f_1(r_1, ..., r_n)$$

.

.

$$r_n = f_n(r_1, ..., r_n)$$

The LFP is guaranteed to exist since the functions f_i are all monotone in the case of Horn clauses.

Algorithm 3 shows the naive evaluation procedure for this set of equations, while algorithm 4 shows the semi-naive evaluation procedure.

Algorithms 1-4 are basically relational algebra programs that execute against the set of base relations. That is, they compute the tuples of the derived relations given the base relations. The efficiency of this program is strongly dependent upon the interface to the DBMS. If the DBMS interface is relational algebra, the above algorithms must

$$r_1^0 = \phi;$$

.

.

$$r_n^0 = \phi;$$

repeat

$$r_1^{j+1} = f_1(r_1^j, ..., r_n^j);$$

.

.

$$r_n^{j+1} = f_n(r_1^j, ..., r_n^j);$$

until $(r_i^{j+1} - r_i^j = \phi)$ for $i = 1, ..., n;$

Algorithm 3: Naive Evaluation of a Set of Recursive Equations

be executed as application programs since relational algebra cannot express least fixed point queries [Aho79]. During each iteration several temporary tables would have to be created and dropped. Also, checking for termination of the iteration involves set difference, a costly operation. (In the next chapter, we describe naive and semi-naive LFP evaluation algorithms with relational algebra as the DBMS interface. We have used relational algebra as the DBMS interface in the VLPDF demonstration testbed, since this testbed is built on top of an existing relational DBMS).

On the other hand, if the DBMS interface allows expressing the above system of recursive equations, the DBMS can better optimize the LFP computation, avoiding these overheads. Also, the DBMS may be able to optimize certain forms of these recursive equations (e.g., transitive closure) better than others. The issues then are: What are these forms? What is the best way of implementing them? Which of these forms do we include in the DBMS interface? How should this be done? These issues significantly affect the efficiency of LFP computation, and thereby D/KB query processing performance. Therefore, the DBMS interface is a very critical design parameter for the D/KBMS architecture.

Another way of improving the efficiency of LFP computation is to restrict the search space, i.e., to select only those tuples of the relations r_i, $i = 1, 2, ..., n$, that are needed for the computation. In the next section, we discuss various optimization techniques that have been proposed to restrict the search space.

$$r_1^0 = \phi;$$

.

.

$$r_n^0 = \phi;$$

$$\delta r_1^0 = f_1(\phi);$$

.

.

$$\delta r_n^0 = f_n(\phi);$$

while some $\delta r_i^j \neq \phi$ **do** {

$$r_1^{j+1} = r_1^j \cup \delta r_1^j;$$

.

.

$$r_n^{j+1} = r_n^j \cup \delta r_n^j;$$

$$\delta r_1^{j+1} = f_1(r_1^{j+1}, ..., r_n^{j+1}) - f_1(r_1^j, ..., r_n^j);$$

.

.

$$\delta r_n^{j+1} = f_n(r_1^{j+1}, ..., r_n^{j+1}) - f_n(r_1^j, ..., r_n^j);$$

}

Algorithm 4: Semi-naive Evaluation of a Set of Recursive Equations

4.7. Optimization

In this section, we discuss optimization strategies meant for use with bottom-up evaluation. These strategies improve the efficiency of LFP computation. Several strategies have been proposed, e.g., magic sets [Banc86 ...], supplementary magic sets [Sacc86], counting and supplementary counting [Sacc86.......]. The main idea behind these strategies is the use of *sideways information passing* to restrict the computation to tuples that are related to the query. Beeri and Ramakrishnan [Beer86] have developed a

uniform framework to describe and compare these strategies and to understand the basic ideas that are common to them. They first formalize the notion of sideways information passing. Then they describe four strategies in terms of this formalism. They call these strategies generalized magic sets, generalized supplementary magic sets, generalized counting, and generalized supplementary counting. We will describe their formalism and the generalized magic sets strategy later in this section. But first we will give a flavor of sideways information passing and optimization.

Given bindings for some variables of a predicate, we can evaluate the predicate with these bindings. This evaluation generates bindings for the other variables of the predicate. These new bindings can be passed to another predicate in the same rule to restrict the computation for that predicate.

Example: Given the rules

$$R_1: \quad r(X, Y) \leftarrow s(X, Z), b_1(Z, Y).$$
$$R_2: \quad r(X, Y) \leftarrow b_2(X, Y).$$

$$R_3: \quad s(X, Y) \leftarrow b_3(X, Y).$$

the query

$$R_4: \quad query(X) \leftarrow r(X, ``a").$$

gives a binding for the second argument of r. Since the second argument of r is the same as the second argument of b_1, this binding restricts the computation of b_1. After evaluating b_1, we get bindings for Z, which in turn can be passed to s to restrict the computation of s. []

In terms of relational algebra, sideways information passing corresponds to pushing selections down a relational algebra tree.

Example: Evaluating the above query corresponds to evaluating the following relational algebra expression

$$\pi_1 \, \sigma_{2=``a"} \, (b_2 \cup (\pi_{1,4} \, b_3 \underset{2=3}{\bowtie} b_1))$$

Use of sideways information passing corresponds to evaluating

$$\pi_1 \, \sigma_{2=``a"} b_2 \cup (\pi_{1,4} \, b_3 \underset{2=3}{\bowtie} (\sigma_{2=`a`} b_1)))$$

where the $\sigma_{2=``a"}$ selection has been pushed down to restrict the number of tuples of b_2 and b_1. []

As seen in the above example, sideways information passing is easily accomplished for nonrecursive rules. However, for recursive rules the situation is more complicated. We cannot simply push the selection down since we run the risk of losing result tuples during the LFP computation. Consider the following data/knowledge base and query

$ancestor(X, Y) \leftarrow parent(X, Y).$

$ancestor(X, Y) \leftarrow parent(X, Z), ancestor(Z, Y).$

$parent("john", "jack"), parent("john", "mary"),$

$parent("jack", "evan"), parent("jack", "ellen"),$

$parent("mary", "brian"), parent("mary", "ann"),$

$parent("joe", "charles"), parent("joe", "diana"),$

$parent("charles", "ben"), parent("charles", "jan")$

$query(X) \leftarrow ancestor("john", X).$

Evaluating this query is equivalent to evaluating

$$\sigma_{1="john"} \text{LFP}(ancestor = parent \cup \pi_{1,4}(parent \underset{2=3}{\bowtie} ancestor))$$

The LFP computation will yield the following tuples for $ancestor$.

$ancestor("john", "jack"), ancestor("john", "mary"), ancestor("john", "evan"),$

$ancestor("john", "ellen"), ancestor("john", "brian"), ancestor("john", "ann")$

$ancestor("jack", "evan"), ancestor("jack", "ellen")$

$ancestor("mary", "brian"), ancestor("mary", "ann")$

$ancestor("joe", "charles"), ancestor("joe", "diana"), ancestor("joe", "ben"),$

$ancestor("joe", "jan"), ancestor("charles", "ben"), ancestor("charles", "jan")$

Applying the selection $\sigma_{1="john"}$ to this relation then yields the following results for the query

$ancestor("john", "jack"), ancestor("john", "mary"), ancestor("john", "evan"),$

$ancestor("john", "ellen"), ancestor("john", "brian"), ancestor("john", "ann")$

Let us see what happens if we try to push the $\sigma_{1="john"}$ selection through the LFP operation down to the parent relation. We would then be evaluating the following LFP expression

$$\text{LFP}(ancestor = \sigma_{1="john"} parent \cup \pi_{1,4}((\sigma_{1="john"} parent) \underset{2=3}{\bowtie} ancestor))$$

This LFP computation yields the following tuples

$ancestor("john", "jack"), ancestor("john", "mary")$

which is obviously not the correct answer to the query.

The source of the problem is that selection commutes across a cartesian product $A \times B$ only if it applies either entirely to A or entirely to B. Therefore, we cannot apply the selection to parent prior to evaluating the join of *parent* and *ancestor*.

What we need is a way of determining prior to the LFP computation all the relevant tuples of the *parent* relation that will be needed. Let us first introduce some definitions to clarify the notion of relevant facts. These definitions are from [Banc86]. A fact $p(a)$ is *relevant* to a query q if $p(a)$ is reachable from $q(b)$ for some b in the answer set. A *sufficient set of relevant facts* for a query is a set of facts such that replacing the extensional knowledge base with this set of facts gives the same answer to the query. A *set of potentially relevant facts* is a superset of the set of relevant facts. A set of potentially relevant facts is *valid* if it contains a sufficient set of relevant facts.

In general, it is impossible to find all the relevant facts for a query without expending as much effort as is needed to evaluate the query itself. The optimization strategies for bottom-up evaluation, therefore, compute only a valid set of potentially relevant facts. The major metric for evaluating an optimization strategy is the difference between the size of this set and that of the sufficient set of relevant facts contained in it.

The optimization strategies mentioned at the beginning of this section are all rule rewriting strategies. The new set of rules is equivalent to the original set but its LFP computation is more efficient. As we mentioned before, [Beer86] have developed a unified framework for describing and comparing these strategies. They have also developed four strategies — generalized magic sets, generalized supplementary magic sets, generalized counting, generalized supplementary counting — in terms of this framework. Here, we will describe the generalized magic sets strategy. But first, we need to describe the formal meaning of sideways information passing.

4.7.1. Sideways Information Passing

Following [Beer86], let r be a rule with head predicate h. If a predicate occurs more than once in the body of r, we number its occurrences. Let $P(r)$ denote the set that contains the head predicate and the predicates occurring in the body of r.

A sideways information passing strategy, called a *sip*, for a rule r is a labeled directed acyclic graph where:

(i) Each node is either a member of $P(r)$ or a subset of $P(r)$.

(ii) Each arc is of the form $N \to p$ where N is a subset of $P(r)$, and p is a member of $P(r)$. Arcs are such that there are no cycles in the sip.

(iii) Each arc has a label χ, which is a set of variables each of which appears in some member of N.

Since the sip is acyclic there exists a total ordering of the predicates in $P(r)$ such that for each arc, all members of its tail appear before its head. The predicates in the rule are evaluated according to this total order. The evaluation is done as follows. For each arc $N_i \to p$ with label χ_j entering p, we compute the join of the predicates in N_i (some

arguments of these predicates may be bound to constants). These values are passed to the predicate p and are used to restrict its computation.

Example:

$$R_1: \quad ancestor(X, Y) - parent(X, Y).$$

$$R_2: \quad ancestor(X, Y) - parent(X, Z), ancestor.1(Z, Y).$$

$$query(X) - ancestor("john", X).$$

We have numbered the occurrence of *ancestor* in the body of R_2 to distinguish it from the head. The natural way to use R_2 is to evaluate predicates in the indicated order, passing bindings from one predicate to another. This strategy can be represented by the following sip

$$\{ancestor, parent\} \rightarrow ancestor.1, \quad \chi = Z \quad []$$

Example

$$R_1: \quad sg(X, Y) - flat(X, Y).$$

$$R_2: \quad sg(X, Y) - up(X, Z_1), sg.1(Z_1, Z_2), flat(Z_2, Z_3), sg.2(Z_3, Z_4), down(Z_4, Y).$$

$$query(X) - sg("john", X).$$

Here also it is natural to evaluate the predicates in R_2 in the indicated order. We show two possible sips below.

$$sip\,1: \quad \{sg, up\} \rightarrow sg.1 \quad \chi = Z_1$$

$$sip\,2: \quad \{sg, up\} \rightarrow sg.1 \quad \chi = Z_1$$
$$\{sg, up, sg.1, flat\} \rightarrow sg.2 \quad \chi = Z_3$$

In $sip\,1$, values of Z_1 are used to restrict the evaluation of $sg.1$. In $sip\,2$, in addition, values of Z_3 are used to restrict evaluation of $sg.2$. []

Beeri and Ramakrishnan in their paper just describe what a sip is. They do not present an algorithm for generating a sip, given a rule and a query. We have developed such an algorithm, which we will describe in the next section as part of the adorned rule set generation algorithm.

4.7.2. Adorned Rule Set

The first step in rewriting the set of rules into a more efficient form is to generate the *adorned rule set*. To understand this step, we need to introduce some definitions. An *adornment* of a predicate with arity n is a sequence of length n of b's and f's [Ullm85]. The adornment indicates which arguments are to be considered as bound

during the evaluation of a predicate and which will get values as a result of the evaluation. The bound arguments are denoted by b, while those that will get values as a result of the evaluation are denoted by f in the adornment.

Example: The sequence $fbbf$ is an adornment of a predicate with arity 4. []

Example: The adornment $fbbf$ indicates that the second and third arguments are to be considered bound and that the first and fourth arguments will get values as a result of the evaluation. []

An *adorned predicate* is a predicate augmented with its adornment, e.g., p^{fbbf}. An *adorned rule* is a rule where (1) the head predicate is adorned, and (2) the derived predicates in the body are adorned (thus, only derived predicates have an adorned version).

Example: $p^{bf}(X, Y) \leftarrow b(X, Z), s^{fb}(Y, Z)$, where b is a base predicate and s a derived predicate. []

We remarked earlier that informally a sip describes how bindings are passed between predicates. An adorned rule formalizes this description. For example, the above adorned rule says that to evaluate p with X bound and Y free, we evaluate b with X bound to get values for Z. These values values are then used to evaluate s with Z bound and Y free.

The process of generating the adorned rule set starts with the query. The constants in the query define adornments for the query predicates. For example, the query $query(X) \leftarrow ancestor("john", X)$ defines the adornment bf for *ancestor*.

For each adornment a of a derived predicate p, we create an adorned predicate p^{a}. We then generate adorned rules defining p^{a}. The adorned rules are generated using the original rules defining p. We use the following recursive procedure to generate the adorned rules defining an adorned predicate p^{a}. For each rule r with head p,

(i) Generate a sip for this rule corresponding to the adornment a using the sip generation algorithm described in section 4.7.2.1.

(ii) Generate a new rule with head p^{a}.

(iii) Replace each occurrence of a derived predicate d in the body by its adorned version. We do this as follows. Let χ denote the union of the labels of all arcs entering d in the sip. If there is no arc entering d, χ is set to empty. The adornment for the predicate occurrence d is a_d where a variable of d is bound in a_d if it appears in χ. If χ is empty, the adornment a_d contains only f's.

(iv) Generate adorned rules defining d^{a_d} if they have not been generated before. []

4.7.2.1. Sip Generation Algorithm

As we mentioned before, Beeri and Ramakrishnan in their paper only describe what a sip is, but do not say how the sip is to be generated. We have developed a sip generation algorithm, which we describe in this section.

1. Initialize the set of sip arcs to empty.

2. Generate the *rule-pred graph*. The nodes of this graph are predicates. The node corresponding to the head predicate is called *head*. An edge between two nodes means that the corresponding predicates have variables in common. The edges are labeled to denote the common variables.

3. For each derived predicate node p, find all simple paths from *head* to p. A simple path between two nodes p and q is a path $p, p_1, ..., p_n, q$, where each node appears only once. Each such path is a potential arc in the sip. For example, the path $\{head, p_1, p_2, ..., p_n, p\}$ corresponds to the arc $\{head, p_1, p_2, ..., p_n\} \rightarrow p$. The label χ for this arc is obtained as follows. For each node q in the tail of the arc, if there is an edge between q and p in the rule-pred graph, we add the variables denoted by this edge to χ. This step enumerates for each derived predicate p all the possible arcs with p as the tail.

4. For each derived predicate, order the potential arcs in descending order of the number of predicates in the tail. Thus, $\{head, p_1, p_2\} \rightarrow p$ comes before $\{head, q\} \rightarrow p$.

5. Order the derived predicates in descending order of the sum of the number of predicates in the tail. Thus, if p has arcs $\{head, p_1, p_2\} \rightarrow p$ and $\{head, q\} \rightarrow p$ and if s has one arc $\{head, p_1, p_2, p_3, p_4, p_5, p_6\} \rightarrow s$, then s comes before p.

6. For each derived predicate p (as per the order of step 5), add arcs $\{head, p_1, p_2, ..., p_n\} \rightarrow p$ (as per the order of step 4) if (1) the edge between *head* and p_1 contains a variable that is bound in the adornment a, (2) adding the arc does not cause a cycle in the sip, and (3) adding the arc causes a new variable of p to appear in the label χ. []

The process of determining the set of arcs for sip is exponential in the number of derived predicates in the body of the rule. Steps 4 and 5 are heuristics that tend to favor arcs with more predicates in their tail. As we shall see when we describe the generalized magic sets algorithm, this will keep the size of the potential set of relevant facts closer to the actual set of relevant facts.

Another feature of this algorithm is that each arc in the sip will have *head* in its tail. This feature makes possible an important optimization in the generalized magic sets algorithm. See [Beer86] for details.

Example:

R_1: $ancestor(X, Y) \leftarrow parent(X, Y)$.

R_2: $ancestor(X, Y) \leftarrow parent(X, Z), ancestor.1(Z, Y)$.

$query(X) \leftarrow ancestor("john", X)$.

The query gives the adornment bf for *ancestor*. The sip for R_2 for this adornment is $\{ancestor, parent\} \rightarrow ancestor.1$, with label $\chi = Z$. The adorned rules are

$ancestor^{bf}(X, Y) \leftarrow parent(X, Y)$.

$$ancestor^{bf}(X, Y) - parent(X, Z), ancestor.1^{bf}(Z, Y).$$

$$query^{f}(X) - ancestor^{bf}("john", X). \quad []$$

Example:

$$sg(X, Y) - flat(X, Y).$$
$$sg(X, Y) - up(X, Z_1), sg.1(Z_1, Z_2), flat(Z_2, Z_3), sg.2(Z_3, Z_4), down(Z_4, Y).$$

$$query(X) - sg("john", X).$$

Using the sip

$$\{sg, up\} - sg.1 \quad \chi = Z_1$$
$$\{sg, up, sg.1, flat\} - sg.2 \quad \chi = Z_3$$

we get the following adorned rules

$$sg^{bf}(X, Y) - flat(X, Y).$$
$$sg^{bf}(X, Y) - up(X, Z_1), sg.1^{bf}(Z_1, Z_2), flat(Z_2, Z_3), sg.2^{bf}(Z_3, Z_4), down(Z_4, Y).$$

$$query^{f}(X) - sg^{bf}("john", X). \quad []$$

Beeri and Ramakrishnan give a formal proof that for a given query, the adorned rule set generated using the above algorithm is equivalent to the original rule set.

4.7.3. Generalized Magic Sets

The second (and final) step in the rule rewriting transformation is to define additional predicates that compute the values that are passed between predicates according to the chosen sip. Each of the original rules is modified by including these additional predicates in the rule body. This ensures that a rule is evaluated only when the values for these additional predicates are available, thereby restricting the search space. The additional predicates are called *magic predicates* and the values they compute are called *magic sets*. The rules defining magic predicates are called *magic rules*. The original rules modified to include magic predicates in their body are called *modified rules*.

We describe the transformation below.

(i) For each adorned predicate, p^{a}, we create a new predicate, $magic\_p^{a}$. The arity of $magic\_p^{a}$ is equal to the number of b's in the adornment a. The arguments of $magic\_p^{a}$ correspond to the bound arguments in the adornment.

(ii) For each adorned rule r and each occurrence of an adorned predicate p^{a} in its body, we generate a magic rule defining $magic\_p^{a}$. Let χ denote an argument list. Then χ^{f} (respectively, χ^{b}) denotes χ with all arguments that are bound

(respectively, free) in the adornment a deleted. Let the adorned rule r be defined as follows

$$r: \quad p^a(\chi) \leftarrow q_1^{a_1}(\chi_1),\ q_2^{a_2}(\chi_2),\ ...,\ q_n^{a_n}(\chi_n).$$

Here a, a_1, ..., a_n denote adornments. The q_i's can be either base or derived predicates. If q_i is a base predicate, its adornment a_i will be understood to be empty.

Let s_r be the chosen sip for this rule. The predicates q_i are assumed to be ordered according to the total order imposed by s_r. That is, predicates participating in the sip precede those that do not and for each arc in s_r, the predicates in the tail precede the predicate at the head.

Consider q_i. Either there is only one arc $N \rightarrow q_i$ in the sip s_r, or there are many. In the former case, we generate the magic rule as follows. The head of this rule is $magic\_q_i^{a_i}(\chi_i^b)$. For each q_j, $j < i$, add $q_j^{a_j}(\chi_j)$ to the body of the magic rule. Finally, add $magic\_p^a(\chi^b)$ to the body.

If there are several arcs entering q_i, we proceed as follows. For each arc $N_j \rightarrow q_i$ with label χ_j, we define a rule with head $label\_q_i(\chi_j)$. The body of this rule is generated as described in the previous paragraph. The magic rule is then defined as a rule with $magic\_q_i^{a_i}(\chi_i^b)$ as head and $label\_q_i(\chi_j)$ for all j as the body.

(iii) Modify each adorned rule by adding a magic predicate to its body. The magic predicate corresponds to the head predicate.

(iv) Add the fact $magic\_q^{a_q}(\chi_q^b)$ to the set of magic rules, where the query is q with adornment a_q and argument list χ_q. []

Example:

$$ancestor^{bf}(X,\ Y) \leftarrow parent(X,\ Y).$$
$$ancestor^{bf}(X,\ Y) \leftarrow parent(X,\ Z),\ ancestor^{bf}(Z,\ Y).$$

$$query(X) \leftarrow ancestor^{bf}("john",\ X).$$

The magic rules are

$$magic\_ancestor^{bf}(Z) \leftarrow magic\_ancestor^{bf}(X),\ parent(X,\ Z).$$
$$magic\_ancestor^{bf}("john").$$

The modified rules are

$$ancestor^{bf}(X,\ Y) \leftarrow magic\_ancestor^{bf}(X),\ parent(X,\ Y).$$

$$ancestor^{bf}(X, Y) \leftarrow magic\_ancestor^{bf}(X), parent(X, Z), ancestor^{bf}(Z, Y). \quad []$$

We note the following points from this example.

(1) For the *parent* relation mentioned in a previous example, evaluating the magic rules yields the following tuples for $magic\_ancestor^{bf}$

$$magic\_ancestor^{bf}(\text{"john"}), magic\_ancestor^{bf}(\text{"jack"}), magic\_ancestor^{bf}(\text{"mary"}),$$

$$magic\_ancestor^{bf}(\text{"evan"}), magic\_ancestor^{bf}(\text{"ellen"}), magic\_ancestor^{bf}(\text{"brian"}),$$

$$magic\_ancestor^{bf}(\text{"ann"})$$

These tuples constitute the magic set for $magic\_ancestor^{bf}$.

(2) The magic set is a projection on the first column of the set of relevant tuples of the *parent* relation.

(3) The join of $magic\_ancestor^{bf}$ and *parent* yields all the relevant tuples of *parent* needed to solve the query.

(4) Adding $magic\_ancestor^{bf}$ to the body of the modified rules forces this join to be evaluated. This also ensures that the modified rules will be evaluated only after the magic set has been computed.

(5) The modified rules are evaluated using only the relevant tuples of the *parent* relation thereby reducing the search space. As a matter of fact, the modified rules can be written as

$$ancestor^{bf}(X, Y) \leftarrow relevant\_parent(X, Y).$$
$$ancestor^{bf}(X, Y) \leftarrow relevant\_parent(X, Z), ancestor^{bf}(Z, Y).$$

$$relevant\_parent(X, Y) \leftarrow magic\_ancestor^{bf}(X), parent(X, Y).$$

Beeri and Ramakrishnan give a formal proof that the set of magic rules and modified rules is equivalent to the set of adorned rules.

4.8. Conclusions

This chapter described the important concepts pertaining to D/KB query processing. In our work, the data/knowledge base is considered to be a set of Horn clauses and schemas. Definitions pertaining to the structure and composition of such a D/KB were given. Recursive query processing was seen to be a key concept differentiating D/KB query processing from traditional database query processing. The concepts of recursive and nonrecursive predicates, recursive and nonrecursive rules, reachability, mutual recursion, and cliques were described. Two basic strategies for Horn clause query evaluation — top-down and bottom-up — evaluation were then described. Top-down strategies were seen to be more efficient but more complex and harder to implement. Bottom-up strategies were seen to be simpler and easy to implement but did a lot of useless work. Bottom-up evaluation of nonrecursive predicates was shown to be

accomplished via a straightforward compilation to relational algebra, while that of recursive predicates involved evaluating the LFP of a set of recursive equations. Two basic strategies for bottom-up LFP computation — naive and semi-naive evaluation — were then described. Naive evaluation was seen to be more inefficient as it recomputed tuples computed during previous iterations. Semi-naive evaluation was seen to avoid much redundant work by computing the differential of the right hand side of the recursive equations. Finally, the concepts relating to D/KB query optimization were described. Sideways information passing to restrict the search space to the relevant base relation tuples and rewriting the rules in the D/KB to an equivalent form whose LFP computation is more efficient were seen to be the basic ideas behind D/KB query optimization strategies. A novel sideways information passing algorithm was described.

CHAPTER 5

Transitive Closure Algorithms

One of the difficult problems in the design of a D/KBMS is how to evaluate recursive queries efficiently. In general, the solution to a recursive query cannot be expressed as a finite relational algebraic expression and therefore cannot be evaluated directly by a conventional relational database system [Aho79].

Among the large family of recursive queries, the *transitive closure query*, a query whose processing requires the computation of the transitive closure of a database relation, is a very important class of recursive queries. They are important because (1) a large number of recursive queries can be expressed using transitive closures [Agra87, Rose86]. (2) most application problems involving recursive queries which we can see now are actually transitive closure queries, and (3) efficient processing of transitive closure queries will provide a sound base for solving more complicated recursive queries. It is thus not surprising that much effort has been devoted to the efficient computation of the transitive closure of database relations recently [Ioan86, Rose86, Vald86]. There is even a tendency to extend relational algebra to include the operation of transitive closure in relational database management systems [Agra87].

This chapter describes our evaluation of algorithms for computing the transitive closure of a database relation. The results of this evaluation appears in [Lu87]. Based on this investigation, we concluded that it is possible to further optimize transitive closure processing. This led us to develop new strategies for this problem, which we also describe in this chapter.

The chapter is organized as follows. Section 5.1 presents definitions and background relating to transitive closure. Section 5.2 presents four algorithms for computing the transitive closure of a database relation: the Brute Force and Logarithmic iterative algorithms [Vald86], Warshall's algorithm, and Warren's algorithm. Section 5.3 describes two implementations of the Logarithmic algorithm and one implementation of Warren's algorithm, and analyzes their performance. Section 5.4 gives the results of our performance comparison. Section 5.5 presents conclusions from the evaluation of these algorithms. Section 5.6 presents two new transitive closure evaluation strategies.

5.1. Definitions and Background

If $R0(a,b)$ is a database relation, its transitive closure $R = R0^+$ is defined by

$$R = R0^+ = \bigcup_{i \geq 1} R^i \tag{5.1}$$

where R^i denotes the ith power of $R0$: $R^1 = R0$ and $R^n = R^{n-1} \bigcirc R$ for $n > 1$. The composition operator $\bigcirc$ on the two binary relations R and S is defined by

$$R \bigcirc S = \{(x,z) \mid \exists y \ (x,y) \in R \wedge (y,z) \in S\}$$

Using relational algebra, this composition can be expressed as

$$R \bigcirc S = \pi_{R.a, S.b} (\underset{R.b=S.a}{R \ \text{join} \ S})$$

Graphically, relation $R0$ can be represented as a directed graph $G(V,E)$, where a node $a \in V$ represents a domain value of $a \in \{R0.A, R0.B\}$, and a directed edge e in E, $a \to b$, represents a tuple (a,b) in the relation $R0$. Then, a node pair (x,y) is in the transitive closure of $R0$, R (or $R0^{+}$) whenever there is a path of nonzero length from x to y. The longest path length, that is, the largest number of edges comprising a path, is sometimes referred to as the *depth* of the transitive closure. We will follow the same convention in our discussion.

More formally, the transitive closure of relation $R0$ represents the derived relation R defined by the following Horn clauses:

$$R(x,y) \leftarrow R0(x,y).$$
$$R(x,y) \leftarrow R(x,z), \ R0(z,y).$$

The transitive closure can be used to evaluate more complex recursive queries. Consider for example the *EMP_SAL* and *EMP_MGR* relations shown in figure 5.1. Note that not all employees have managers. Consider the query, "For each manager, list the names of his subordinates (direct or indirect) and their total salary." This query can be expressed in SQL (augmented with the transitive closure function) as shown in figure 5.2.

Equation (5.1) is the basis for several iterative transitive closure algorithms [Vald86]. These algorithms compute successive approximations to the right hand side of equation (5.1) until convergence is obtained.

Several years ago, Warshall described an essentially different algorithm for computing the transitive closure of a relation [Wars62]. Warshall's algorithm was originally designed to compute the transitive closure of a relation represented as an adjacency matrix. An adjacency matrix is a two dimensional Boolean array M, where $M(x,y)=$**true** whenever $(x,y) \in R$, otherwise $M(x,y)=$**false**. This algorithm has the remarkable property that it can compute the transitive closure in a single pass over the matrix, in contrast to the indefinite number required by iterative algorithms. Warren subsequently modified this algorithm to give it better performance in a virtual memory environment [Warr75]. We have developed an implementation of Warren's algorithm in which the relation is represented as a set of tuples, as is usual in relational database systems. In this form, the algorithm can be integrated into a relational database system for the purpose of evaluating recursive queries.

EMP_SAL	
Ename	Salary
R. Smith	30k
A. Bailey	40k
B. Sullivan	40k
N. Johnson	45k
R. Elliott	35k
K. Doty	40k
C. Shaffer	45k
T. Benton	50k
J. Kennedy	43k
N. Sibell	45k

EMP_MGR	
Ename	Mname
R. Smith	B. Sullivan
A. Bailey	N. Johnson
B. Sullivan	N. Johnson
N. Johnson	N. Sibell
R. Elliott	B. Sullivan
K. Doty	N. Sibell
C. Shaffer	T. Benton
J. Kennedy	N. Sibell
N. Sibell	T. Benton

Figure 5.1. Relations for example recursive query.

5.2. Algorithms for Transitive Closure

In the following descriptions of transitive closure algorithms, R is the binary source relation with attributes A and B. T is the result relation and has the same attribute names.

5.2.1. Brute Force Iterative Algorithm

This version of the Brute Force algorithm works directly on the base relations; Valduriez and Boral also present a version of the algorithm that uses a join index to improve processing speed [Vald86]. We have modified the algorithm to work properly with cyclic as well as acyclic relations, for a fair comparison with Warshall's and Warren's algorithms. The Brute Force algorithm can be expressed as follows:

$$T := R;$$
$$R_\Delta := R;$$

```
INSERT INTO T(Ename, Mname)
SELECT *
FROM TRANSITIVE_CLOSURE(EMP_MGR);

INSERT INTO U(Mname, Ename, Salary)
SELECT T.Mname, T.Ename, EMP_SAL.Salary
FROM T, EMP_SAL
WHERE T.Ename = EMP_SAL.Ename

SELECT Mname, Ename, SUM(Salary)
FROM U
GROUP BY Mname
```

Figure 5.2. Implementation of recursive query.

```
while R_Δ ≠ ∅ do
     begin
     R_Δ := R_Δ·R ;
     R_Δ := R_Δ - T ;
     T := R_Δ ∪ T ;
     end
```

Note that the set union $R_\Delta \cup T$ is a disjoint union. After i iterations of the **while** loop,

$$R_\Delta = R^{i+1} - \bigcup_{1 \le j \le i} R^j$$

$$T = \bigcup_{1 \le j \le i+1} R^j$$

The algorithm terminates when

$$R^{i+1} \subseteq \bigcup_{1 \le j \le i} R^j \tag{5.2}$$

From this it is easy to show that

$$T = \bigcup_{1 \le j \le i+1} R^j = \bigcup_{j \ge 1} R^j = R^+$$

The number of iterations required can be expressed in terms of the directed graph defined by R. Let $\text{paths}(x,y)$ denote the set of paths from x to y in the graph. Let $\text{length}(s)$ denote the length of path s, i.e., the number of edges. Define the quantity p by

$$p = \max_{\substack{z,y \\ \text{paths}(z,y) \neq \varnothing}} \quad \min_{s \in \text{paths}(z,y)} \quad \text{length}(s)$$

It is easy to show that (5.2) holds when $i \geq p$, so the Brute Force algorithm requires p iterations to compute the transitive closure of R.

5.2.2. Logarithmic Algorithm

The following version of Valduriez and Boral's Logarithmic algorithm works with cyclic relations:

$$T := R;$$
$$R_\Delta := R;$$
$$X := R;$$
while $X \neq \varnothing$ **do**
  begin
   $R_\Delta := R_\Delta \cdot R_\Delta;$
   $T_\Delta := T \cdot R_\Delta;$
   $X := R_\Delta - T;$
   $Y := R_\Delta \cup T;$
   $T := Y \cup T_\Delta$
  end

Again, the set union $R_\Delta \cup T$ is a disjoint union. After i iterations,

$$T = \bigcup_{1 \leq j \leq 2^{i-1}-1} R^j$$

$$R_\Delta = R^{2^i}$$

$$X = R^{2^i} - \bigcup_{1 \leq j \leq 2^i-1} R^j$$

The Logarithmic algorithm terminates when

$$R^{2^i} \subseteq \bigcup_{1 \leq j \leq 2^i-1} R^j$$

This occurs when $2^i - 1 \geq p$, i.e., when $i \geq \lg(p+1)$. Therefore the Logarithmic algorithm requires $\lceil \lg(p+1) \rceil$ iterations. Valduriez found that the Logarithmic algorithm generally performed better than the Brute Force algorithm; we will therefore use it as our iterative algorithm in what follows.

5.2.3. Smart algorithms

Ioannidis recently proposed a new set of algorithms, *smart* algorithms, to compute the transitive closure of a relation [Ioan86]. A frame work of optimizing the computation along the same direction as the logarithmic algorithm was provided. According to the smart algorithms, the transitive closure of relation $R0$ is expressed as

$$R^+ = \prod_{k=0}^{x} \left(\sum_{l=0}^{m-1} R0^{l*m^k} \right)$$

With a different m value, different algorithms can be obtained. The logarithmic algorithm is actually the special case of $m = 2$.

$$R^+ = (1+R0)(1+R0^2)(1+R0^4) \cdots$$

5.2.4. Warshall's Algorithm

Warshall proposed a quite different algorithm for computing transitive closure on a binary relation [Wars62]. In Warshall's algorithm, a binary relation R is represented by a boolean adjacency matrix M. With this representation, Warshall's algorithm computes the transitive closure of the relation as follows:

```
for j := 1 to N do
    for i := 1 to N do
        if M(i,j) then
            for k := 1 to N do M(i,k) := M(i,k)∨M(j,k)
```

This algorithm effectively computes the transitive closure in only one pass over M. If the matrix is stored in row major order, and if each row is represented as a string of bits, then the inner loop of this algorithm can be implemented very efficiently using machine instructions that compute the logical-or of two words or bit strings.

Warshall's algorithm works by creating ever shorter paths between two nodes in the directed graph represented by R. Suppose R contains a path from x to y. Before the iteration $j = z$, T contains a path $x, w_1, \ldots, w_n, y$ such that $w_i \geq z$ for all i. If z is in this path, then this iteration creates a similar path in T with z removed. When the algorithm terminates, $(x,y) \in T$. Warshall's paper gives a more formal correctness proof.

5.2.5. Warren's Algorithm

Warren proposed an improvement to Warshall's algorithm in a paging environment if the entire matrix will not fit in memory [War75]. Since Warshall's algorithm scans the matrix by columns and updates it by rows, it will introduce a large number of page faults in a virtual memory environment when the matrix cannot fit in real memory. Warren's algorithm avoids this problem by scanning and updating the matrix by rows. It has two passes instead of Warshall's one pass. However, each pass is over only half of M. Here is Warren's algorithm:

```
for i:=2 to N do
        for j:=1 to i-1 do
            if M(i,j) then
                for k:=` to N do M(i,k):=M(i,k)∨M(j,k)


for i:=1 to N-1 do
        for j:=i+1 to N do
            if M(i,j) then
                for k:=1 to N do M(i,k):=M(i,k)∨M(j,k)
```

A formal correctness proof of the algorithm is given in the original paper [Warr75].

Warren's algorithm can be represented in relational database terms. The domain of R assumed in the original implementation is the range of integers [1, N]. In fact, any finite, totally ordered domain D can be used and the choice of total order $>_D$ is arbitrary. For the large domains commonly occurring in database relations, the adjacency matrix representation of R is impractical; the set of tuples representation is much more compact. To achieve the effect of scanning the adjacency matrix by rows, we maintain the tuples in a sequence sorted by attributes A and B as primary and secondary key, respectively. This gives the following implementation of Warren's algorithm:

```
T := R sorted by attributes <A, B>


for t ∈ T { in sorted order } do
        if t.A >_D t.B then
            insert {t} ·σ_{A = t.B}(T) into T;


for t ∈ T { in sorted order } do
        if t.A <_D t.B then
            insert {t} ·σ_{A = t.B}(T) into T;
```

Tuples inserted into T must be inserted in sorted order. The $<_D$ and $>_D$ comparison operators refer to the total ordering of the domain D. The expression $\{t\} \cdot \sigma_{A = t.B}(T)$ is the composition of the singleton relation consisting of tuple t with the tuples of T whose first attribute matched the second attribute of t. All of the tuples in the result of this composition have t.A as their first attribute value. Inserting these tuples into T, which is clustered on attribute A, is fairly cheap. Performing the selection $\sigma_{A = t.B}(T)$ is also fairly cheap for the same reason.

We investigated a similar implementation of Warshall's algorithm. However, the relation T could not be clustered in such a way that the selection and insertion were cheap. This difficulty reflects the original version of Warshall's algorithm scans the matrix by columns (clustered on B) and updates the matrix by rows (clustered on A). Therefore we chose to evaluate the performance of Warren's algorithm only.

5.3. Implementation of Transitive Closure Algorithms and Their Costs

In this section we discuss the implementation details of the transitive closure algorithms and derive their cost formulas.

5.3.1. Basic Operations and Their Costs

The basic operations involved in the iterative algorithm are binary relation composition, set union and set difference. Composition is a join followed by a project, so we now describe the implementation of join, union, and difference, and derive their costs.

5.3.1.1. Join

For the join operation, we have chosen the hybrid hash join algorithm [DeWi84] because of its superior performance. Hybrid hash join consists of two phases, *partitioning* and *probing*. In the partitioning phase, each of two relations, R and S, are partitioned into a number of disjoint buckets. The bucket sizes of the smaller relation, R, are selected such that a hash table can be constructed for a bucket in memory. After this table is constructed, the tuples in the corresponding bucket of relation S are used to probe the hash table to find matches. In the case that the hash table for the whole relation of R can not fit into memory, there will be more than one iteration to process the buckets. During the partitioning of R, the tuples of one bucket remain in memory to construct the hash table; tuples of other buckets are written back to the disk. Similarly, when relation S is partitioned, the tuples in the first bucket are directly used to probe the hash table in memory and others are written back to the disk. The buckets written back to the disk are read in again to construct and probe the hash table in the later iterations. The partitioning phase requires extra buffers to hold tuples being collected for the various buckets. However we assume that M bytes of memory are available for each hash table, whether or not partitioning is occurring while the hash table is being built. Using the notation in Table 5.1, the cost formulas of the hash-based join of R and S are given in Table 5.2.

5.3.1.2. Difference

The difference operation $R - S$ can be implemented in a similar way as the hybrid hash join. In this case, R and S are partitioned using a hash function on the entire tuple. For each bucket of S, a hash table is constructed in main memory. This table is probed with the tuples of the corresponding bucket of R, and those tuples for which there is no match are added to the result. A notation similar to join selectivity, called *difference selectivity, DS,* is defined as the ratio of the number of tuples in the result relation of the difference operation to the number of tuples in the source relation of R. The cost of this algorithm is shown in Table 5.3.

5.3.1.3. Union

We can use a hash-based algorithm to perform the set union operation $R \bigcup S$ in a similar way. In this algorithm, all the tuples of relation S and the tuples from relation R that don't match with tuples in S are moved to output buffers to form the union

M	Size of the available memory (in bytes)
$\mid R \mid$	Number of pages in relation R
$\mid \mid R \mid \mid$	Number of tuples in relation R
JS	Join selectivity $= \dfrac{\mid \mid R \operatorname{join} S \mid \mid}{\mid \mid R \mid \mid \cdot \mid \mid S \mid \mid}$
US	Union selectivity $= \dfrac{\mid \mid R \bigcup S \mid \mid}{\mid \mid R \mid \mid + \mid \mid S \mid \mid}$
DS	Difference selectivity $= \dfrac{\mid \mid R - S \mid \mid}{\mid \mid R \mid \mid}$
DS_i	Selectivity of Difference in i^{th} iteration
TS	Tuple length (in bytes)
PS	Page size (in bytes)
t_{comp}	Time for comparing two attribute values
t_{move}	Time for moving a binary tuple in memory
t_{hash}	Time for hashing an attribute
t_{read}	Time for reading one page from disk
t_{write}	Time for writing one page to disk

Table 5.1: Notations Used in Cost Formulas.

output. Table 5.4 shows the cost of this algorithm.

5.3.1.4. Combined Union and Difference

The union and difference operations both partition R and S by hashing complete tuples, so we can combine them to compute $R - S$ and $R \bigcup S$ simultaneously. This is useful for both the Brute Force and the Logarithmic algorithm. The total cost of obtaining these two copies, denoted as *Union_Diff(R,S)*, is the sum of the cost of

$$(1) \quad = (|R| + |S|) \cdot t_{read}$$ — Reading of R and S

$$(2) \quad + (||R|| + ||S||) \cdot t_{hash}$$ — Hashing and partitioning of R and S

$$(3) \quad + \frac{(||R|| + ||S||) \cdot TS - M}{TS} \cdot t_{move}$$ — Moving overflow buckets to the output buffers

$$(4) \quad + \frac{(||R|| + ||S||) \cdot TS - M}{PS} \cdot t_{write}$$ — Writing the output buffers to disk

$$(5) \quad + \frac{(||R|| + ||S||) \cdot TS - M}{PS} \cdot t_{read}$$ — Reading those tuples later from disk

$$(6) \quad + \frac{(||R|| + ||S||) \cdot TS - M}{TS} \cdot t_{hash}$$ — Rehashing the tuples to build hash tables

$$(7) \quad + ||R|| \cdot t_{move}$$ — Moving tuples of R to build hash tables

$$(8) \quad + ||S|| \cdot t_{comp}$$ — Probing for a match for all the tuples in S

$$(9) \quad + 2 \cdot ||R|| \cdot ||S|| \cdot JS \cdot t_{move}$$ — Moving the matching tuples to the output buffers

$$(10) \quad + ||R|| \cdot ||S|| \cdot JS \cdot \frac{TS}{PS} \cdot t_{write}$$ — Writing the join output tuples to disk

Table 5.2: Cost $Join(R,S)$ of Hash-Based Join.

$Diff(R,S)$ in table 5.3 and the terms (9) an (10) in the cost table of $Union(R,S)$, Table 5.4. It is shown in Table 5.5.

5.3.2. Cost of Iterative Algorithm

The iterative algorithm (the Logarithmic algorithm described above) requires $\lceil \lg(p+1) \rceil$ iterations. In each iteration there are two joins ($R_\Delta \cdot R_\Delta$ and $T \cdot R_\Delta$), one union ($Y \bigcup T_\Delta$) and one combined union and difference (between R_Δ and T). The cost C_i of iteration i is

(1)-(6)	$= ... + ...$	— the same as (1) - (6) in Table 5.2 of *Join (R, S)*
(7)	$+ \mid\mid S \mid\mid \cdot t_{move}$	— Moving tuples of S to build hash tables for buckets of S
(8)	$+ \mid\mid R \mid\mid \cdot t_{comp}$	— Probing for match tuples in S
(9)	$+ \mid\mid R \mid\mid \cdot DS \cdot t_{move}$	— Moving tuples of R not in S to the output buffers
(10)	$+ \dfrac{\mid\mid R \mid\mid \cdot DS \cdot TS}{PS} t_{write}$	— Writing the output buffers to disk

Table 5.3: Cost $Diff(R,S)$ of Hash-Based Set Difference.

(1)-(8)	$= ... + ...$	— the same as (1) - (8) of *Join (R, S)*
(9)	$- (\mid R \mid + \mid S \mid) \cdot US \cdot t_{move}$	— Moving union output tuples to the output buffers
(10)	$+ \dfrac{(\mid R \mid + \mid S \mid) \cdot US \cdot TS}{PS} t_{write}$	— Writing the output buffers to disk

Table 5.4: Cost $Union(R,S)$ of Hash-Based Set Union.

$$C_i = Join(R^{2^{i-1}}, R^{2^{i-1}})$$

$$+ Join(\bigcup_{1 \le j \le 2^i - 1} R^j, R^{2^i})$$

$$+ Union\_Diff(R^{2^i}, \bigcup_{1 \le i \le 2^i - 1} R^j)$$

$$+ Union(\bigcup_{1 \le i \le 2^i} R^j, \bigcup_{2^i - 1 \le j \le 2^{i-1} - 1} R^j)$$

(1)-(10)	$= ... - ...$	— the same as (1) - (10) of $Diff\ (R,\ S)$
(11)	$- (\ \cdot R\ \cdot\ - \ \cdot\ S\ \cdot\ )\cdot US\cdot t_{move}$	— Moving union output tuples to the output buffers
(12)	$+ \dfrac{(\ \cdot R\ \cdot\ + \ \cdot\ S\ \cdot)\cdot US\cdot TS}{PS}\,t_{write}$	— Writing the output buffers to disk

Table 5.5: Cost $Union\_Diff(R,S)$ of Combined Hash-Based Union and Difference.

The total Cost is

$$C = \sum_{1 \le i \le \lceil \lg(p-1) \rceil} C_i$$

5.3.3. Improved Iterative Algorithm

The preceeding cost computation assumed that each relation is written to disk as a sequential file as it is generated by a join or other relational operation. If this file is used as input to a subsequent operation, it must be partitioned into buckets again. This partitioning involves reading the sequential file and writing all but one bucket back to disk.

We propose here an improved implementation of the iterative algorithm in which each relation, as it is generated, is partitioned in preparation for the next use of the relation. This technique eliminates writing and reading the sequential file. However, it requires more memory for buffers for output relation buckets. As before, we assume that there are relatively few buckets so that the memory available for a hash table is not significantly reduced.

In the improved iterative algorithm, relations T, T_Δ and Y are partitioned on attribute B whenever they are generated. When relation R_Δ is generated, two partitionings are generated: one on attribute A and one on attribute B. Relation X is not partitioned. In fact, it need not be materialized since we don't need to know its exact value, only whether or not it is empty. These partitionings ensure that each relational operation can proceed immediately with bucket-by-bucket processing.

The cost of improved iterative algorithm is computed by modifying the cost for the basic relational operations to reflect partitioning output relations instead of input relations. The cost of partitioning T and T_Δ prior to the first iteration must also be included.

5.3.4. Warren's Algorithm

The physical implementation of our version of Warren's algorithm is based on a particular choice for the total order $>_D$ on the domain D of relation R. Instead of using the normal order relation $>$ on number or chaaracter strings, we define $>_D$ as follows:

$$x >_D y <=> hash(x) > hash(y) \lor (hash(x) = hash(y) \land x > y)$$

That is, two elements of the domain are ordered primary on their hash values and secondarily ordered by the normal ordering for numeric or string attributes. This choice of order allows us to use hashing techniques for sorting, selection and insertion.

Relation T is physically represented as a hash table with a main memory pointer array and disk-resident buckets. The number of buckets is such that one page of each bucket can fit into main memory.

The first step in our version of Warren's algorithm is to create T by sorting relation R on attributes A and B. Given the choice of total order on D, we can sort R by partitioning it into buckets based on the hashed value of attribute A, and then sort each bucket. After partitioning, main memory is ordered as a cache of recently-retrieved buckets.

Table 5.6 shows the cost of partitioning the relation and the cost of sorting the buckets during processing. Here we use R_0, R_1 and R to represent the source relation, the result relation after the first pass and the final result relation, respectively.

(1)	$= R_0 \cdot t_{read}$	— Reading the source binary relation
(2)	$+ R_0 \cdot (t_{hash} + t_{move})$	— Hashing the input relation into partitions
(3)	$+ R_0 \cdot (1 - \dfrac{M}{R_1}) \cdot t_{write}$	— Writing overflow buckets to disk
(4)	$+ R_0 \cdot \dfrac{PS}{TS} \cdot \lg \dfrac{PS}{TS} \cdot (t_{comp} + t_{move})$	— Internal sorting of all the pages

Table 5.6: Cost C_0 of Partitioning R_0 and Sorting R_0 Pages.

In the first pass over T, the buckets are processed in order of hash value. First, the bucket is read into main memory. For each tuple t in the bucket such that $t.A >_D t.B$, the selection $\sigma_{a = t.B}(T)$ is performed using the in-memory hash table index. The condition $t.A >_D t.B$ guarantees that the selected tuples are in the same or an earlier processed bucket. Therefore no disk access is necessary unless the earlier

bucket has been forced out of memory. This is reflected in term (8) of Table 5.7 which gives the cost C_1 for the first pass.

(5)	$= \|R_0\| \cdot (1 - \dfrac{M}{\|R_1\|}) \cdot t_{read}$	— Reading the bucket on disk for processing
(6)	$+ \|R_1\| \cdot t_{comp}$	— Comparing the two attribute values
(7)	$+ (\|R_1\| - \|R_0\|) \cdot (t_{hash} + t_{lookup})$	— Hashing and lookup the directory
(8)	$+ \displaystyle\sum_{i=\frac{M}{TS}+1}^{\|R_1\|} \dfrac{i \cdot TS - M}{i \cdot TS} \cdot t_{read}$	— Reading the data pages not in memory
(9)	$+ (\|R_1\| - \|R_0\|) \cdot \lg \dfrac{PS}{TS} \cdot t_{comp}$	— Locating the tuples in the page
(10)	$+ (\|R_1\| - \|R_0\|) \cdot t_{move}$	— Inserting the generated tuples
(11)	$+ (\|R_1\| - \dfrac{M}{TS}) \cdot t_{move} + (\|R_1\| - \dfrac{M}{PS}) \cdot t_{write}$	— Writing overflow tuples to disk

Table 5.7: Cost C_1 of the First Pass of Warren's Algorithm.

Table 5.8 shows the cost of the second pass over the relation. The formulas are similar to that in Table 5.7, with different relation sizes.

As mentioned above, Warren's algorithm complete the transitive closure computation in two passes, the total cost of the algorithm can therefore obtained as

$$C_w = C_0 + C_1 + C_2$$

5.4. Evaluation

We have chosen four parameters to study the effect of their variation on the performance of the three transitive closure algorithms. These four parameters are 1) the memory size, 2) the source and the result relation sizes, 3) the number of iterations in the iterative algorithms, and 4) the join selectivity We have calculated the total cost of each of the three transitive closure algorithms for various sets of parameter values. The values of the I/O and computation parameters have been fixed in all our experiments.

(12)	$= \|R_1\| \cdot \dfrac{M}{PS} \cdot t_{read}$	— Reading the bucket on disk for processing
(13)	$+ \|R\| \cdot t_{comp}$	— Comparing the two attribute values
(14)	$+ \left(\|R\| - \|R_1\|\right) \cdot (t_{hash} + t_{lookup})$	— Hashing and lookup the directory to find matches
(15)	$+ \displaystyle\sum_{i=\frac{M}{TS}+1}^{\|R\|} \dfrac{i \cdot TS - M}{i \cdot TS} \cdot t_{read}$	— Reading the data pages not in memory
(16)	$+ \left(\|R\| - \|R_1\|\right) \cdot \lg \dfrac{PS}{TS} \cdot t_{comp}$	— Locating the tuples in the page
(17)	$+ \left(\|R\| - \|R_1\|\right) \cdot t_{move}$	— Inserting the found tuple after current processing tuple
(18)	$+ \|R\| \cdot t_{move} + \|R\| \cdot t_{write}$	— Moving output tuples to buffers and writing them to disk

Table 5.8: Cost C_2 of the Second Pass of Warren's Algorithm.

In each experiment, we have varied one parameter value over a range, while keeping the values of the other parameters fixed at some value. The value ranges and the typical values we have used in our evaluation are the following:

Figure 5.3 illustrates the increases in the execution time of the three algorithms as the total available memory size is reduced. Warren's algorithm, as we expected from its original nature, performs very poorly as the memory size is reduced. The actual cross-over value of the memory size at which Warren's algorithm starts performing worse depends on the source and result relation sizes and the values of the other system parameters. However, we can see a significantly better performance from Warren's algorithm when reasonable memory sizes are assumed (e.g. > 2 MB). From figure 5.3, we can observe that for the memory sizes exceeding 4 megabytes (assuming source and result relation sizes of the order of 6 and 8 megabytes) Warren's algorithm performs far better than the iterative algorithms. Since memory sizes of few megabytes are fairly typical in current systems, we can expect Warren's algorithm to perform better than iterative algorithms in most typical applications. We can also observe from the figure that the improved iterative algorithm performs much better than its basic counterpart.

Parameter settings used in evaluation	
Parameter	Values (typical value)
Source relation R_0	500KB-10MB (2MB)
Memory size M	400KB-8MB
Join selectivity JS	$10^{-6} - 10^{-8}(10^{-7})$
Union selectivity, US	1.0 (no duplication)
Difference selectivity, DS	1.0 (no duplication)
Number of iterations, p	1 - 62 (6)
Page size, PS	4K Bytes
Tuple size, TS	8 Bytes
t_{comp}	$3\mu s$
t_{move}	$20\mu s$
t_{hash}	$9\mu s$
t_{lookup}	$6\mu s$
t_{read}	15ms
t_{write}	20ms

Figure 5.4 illustrates the changes in the performance of the transitive closure algorithms as the source relation size is varied, while keeping the memory size fixed. We can observe from this figure a similar behavior as seen from figure 5.3, which is as the memory size available for holding the output of transitive closure becomes limited, the performance of Warren's algorithm deteriorates rapidly.

Figure 5.5 shows the effect of variation in the number of iterations required to compute the transitive closure. Since Warren's algorithm is not iterative, its performance remains the same, while the performance of the iterative algorithms becomes extremely worse compared to warren's algorithm when the number of iterations required becomes large. Therefore, if large number of successors of tuples are involved in computing the transitive closure, it is better to use warren's algorithm.

Figures 5.6 and 5.7 illustrate the changes in the performance of the algorithms as the join selectivity for each iteration is increased and decreased respectively. The performance behavior exhibited in both these figures is the same. In both figures, as the result relation size increases, iterative algorithms show better performance than Warren's algorithm. We can conclude from these figures that the values of join selectivity are not as important as the result relation sizes in affecting the relative performance of the transitive closure algorithms.
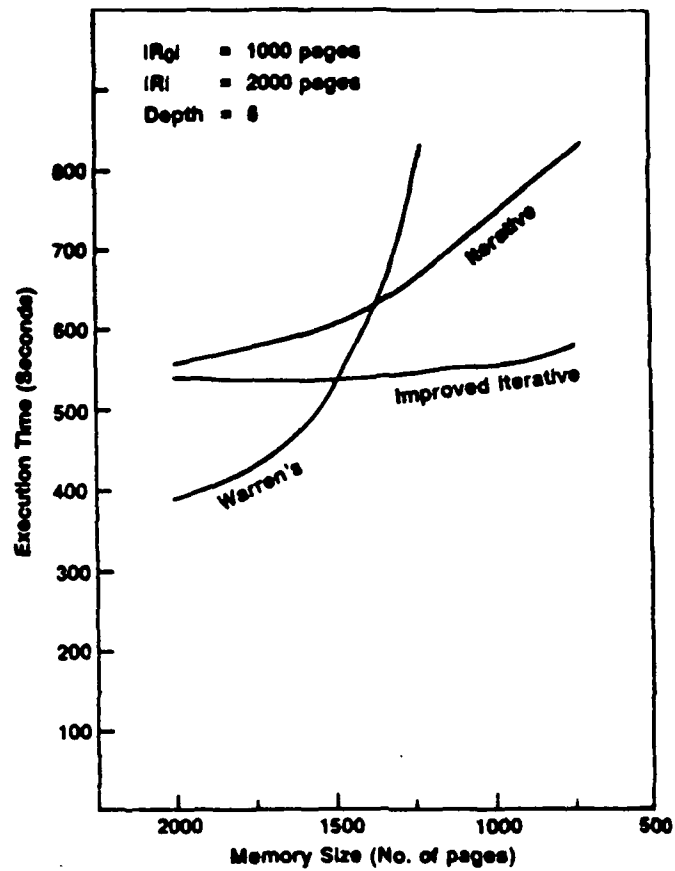
Figure 5.3. Execution Time vs. Memory Size
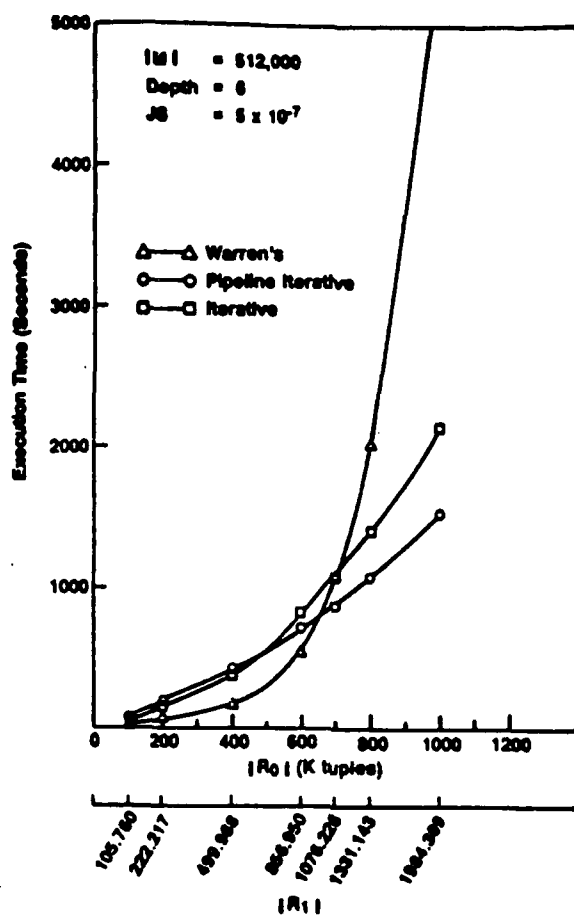
Figure 5.4. Execution Time vs. Relation Size
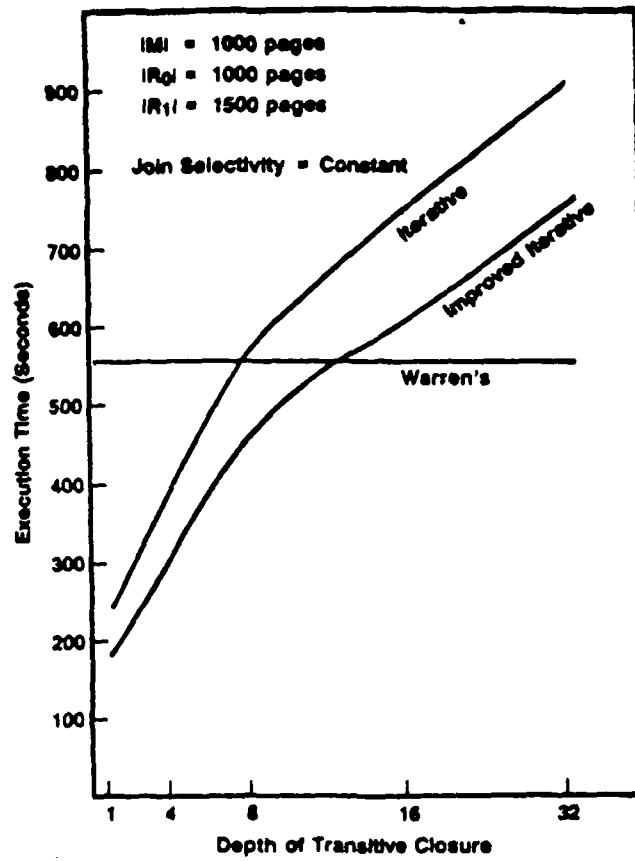
Figure 5.5. Execution Time vs. Depth of Transitive Closure
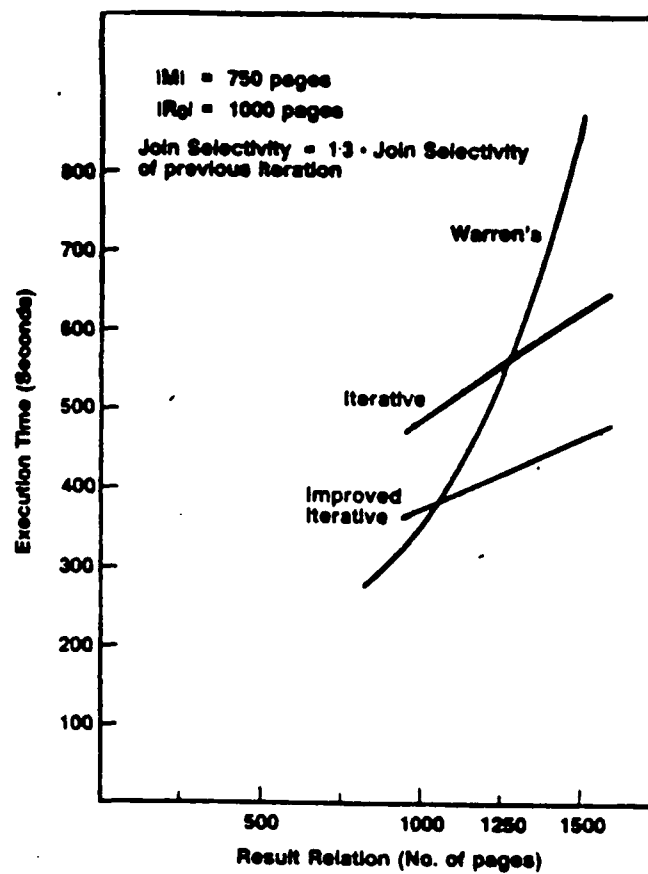
Figure 5.6. Execution Time vs. Increasing Join Selectivity
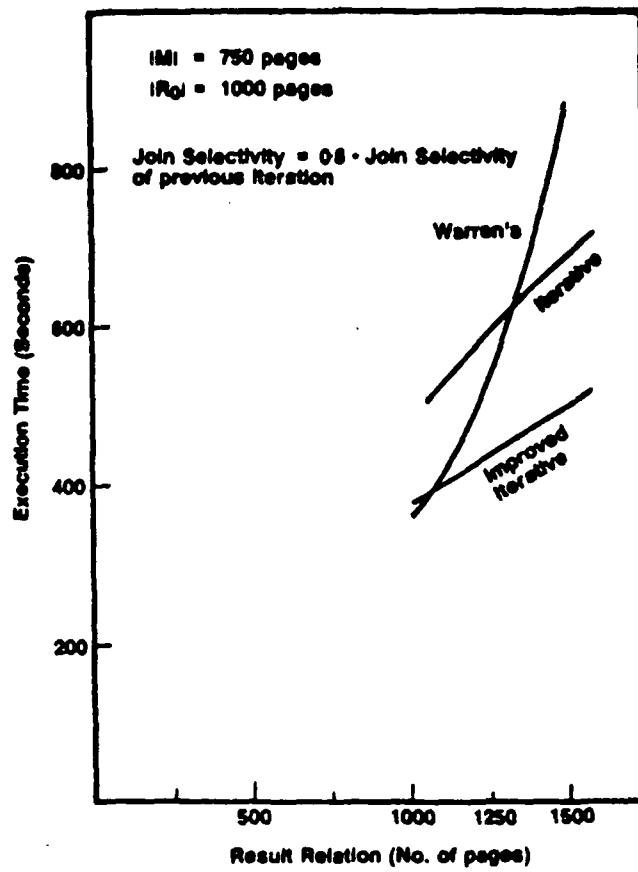
Figure 5.7. Execution Time vs. Decreasing Join Selectivity

5.5. Summary and Discussion

We have presented in this section an adaptation of Warren's algorithm to the relational database environment. It is a non-iterative algorithm and computes the transitive closure of a relation in a depth-first search fashion. This algorithm was compared with a logarithmic iterative algorithm and an improved version of the logarithmic algorithm. The motivation for our study of transitive closure algorithms and their performance is to find some alternative methods in recursive query processing. As we expected, Warren's algorithm, which is basically a simple depth-first search and main memory algorithm, works better in two cases: (1) the relative size of relation is not much larger than the size of available memory, and (2) the path lengths in the transitive closure graph vary greatly. In the second case, the iterative algorithms have to join two whole relations (often very large) iteratively to find a small number of tuples and the total cost increases dramatically. Thus, our recommendation is to implement Warren's algorithm for transitive closure in database systems and let the query optimizer select it adaptively. In the remainder of this section, we briefly present some observations for future work.

Auxiliary Data Structures. In this work, we have not assumed any auxiliary storage structures such as clustered or non-clustered indices and join indices. All operations are applied to the original data. Join indices have been shown to improve the performance of join operations. They also improve the performance of iterative transitive closure algorithms [Vald86]. The reason for this is the size of a join index relation is in general less than the binary relation size. Further investigation of the relative performance improvement of Warren's algorithm resulting from the use of auxiliary data structure is a worthwhile task.

Search Techniques. Depth-first and breadth-first algorithms have been explored extensively to solve the general search and tree traversal problems. Since transitive closure computation is basically a graph search problem, both depth-first and breadth-first algorithms can be employed to compute the transitive closure. Warren's algorithm can be viewed as a depth-first algorithm and iterative algorithms can be viewed as breadth-first algorithms. This analogy can be useful for further research into the application of combined breadth-first and depth-first transitive closure computation techniques as has been suggested in solving other graph search problems. One possible technique is to apply an iterative algorithm a few number of iterations first to find most of the tuples in the transitive closure and then switch to the Warren's algorithm to find the few tuples which can be derived only through longer search paths.

Transitive Closure vs. Fix Point Queries. We concentrated only on the transitive closure algorithms. The algorithms or the evaluation results presented here may not be applicable to performing general least fixpoint operations. Further research into the application of these algorithms to least fixpoint query execution is required.

Restricted Transitive Closure. In general, complete transitive closure is seldom required by applications: a subset of the transitive closure is adequate for answering many queries. The algorithm to handle the restricted transitive closure queries is dependent on the restriction criteria. However, general mechanisms for restricting the

output of each iteration of transitive closure operation and terminating the transitive closure computation after a specified number of iterations are possible. These mechanisms might also be useful in executing general least fixpoint queries.

Multi-processor Transitive Closure Algorithms. Transitive closure is a data-intensive operation. It is possible to partition the task of this very large database processing on to multiple processors and improve the performance of transitive closure computation significantly. For iterative algorithms in multiprocessor environment, the join and union operations in each iteration can be assigned to a separate processor improving the performance through concurrent and pipeline processing. For executing Warren's algorithm using multiple processors, the search of subgraphs starting from different nodes in the graph can be assigned to different processor(s). Another potential area for future work is to design, analyze and evaluate multiprocessor based iterative algorithms and Warren's algorithm.

5.6. New Strategies for Optimizing Transitive Closure Evaluation

In this section, we are going to propose two new strategies that further optimize the computation of the transitive closure of a database relation. As a point of terminology, we refer to our adaptation of Warren's algorithm presented in the previous section as the *recursive algorithm*.

We first assume that the relation we are dealing with is so large that it is impossible to hold all its tuples in main memory. In this case the computation of transitive closure, no matter which algorithm is used, requires a large number of join, union and set difference operations on very large relations. Partitioning a very large relation into smaller disjoint partitions has been proved a reasonable way to dramatically reduce the costs of join operation on large relations [DeWi84]. Both the analysis of the algorithm [Lu87] and the logarithmic algorithm [Vald86] are based on the hash join method. We assume that the same technique is used in our discussion.

5.6.1. Strategy 1: Reduce the Size of R_0

Compared to the naive algorithm, the semi-naive algorithm focus on eliminating the duplication of computation by only using the newly generated tuples as one of the source relations of the join in the next iteration. However, none of the previous algorithms tried to reduce the size of another source relation in the join operation, relation R_0. Since relation R_0 is used in each iteration, its size perhaps has more influence on the performance of the transitive closure algorithms.

Our first optimization strategy is to eliminate dynamically those tuples from relation R_0 that will not generate tuples in the result relation in the later iterations. The example in figure 5.8 is used to explain the strategy.

R_0 consists of 13 tuples. For the semi-naive algorithm, the first iteration joins R_0 with R_0 and generates $\Delta R_1 = R_0 \bigcirc R_0$, which consists of 12 tuples. Traditionally, the second iteration will join ΔR_1 with R_0 again to generate $\Delta R_2 = \Delta R_1 \bigcirc R_0$. However, if we examine the join process, we can find that some tuples in R_0 will never introduce

Iteration 0		Iteration 1		Iteration 2	
R_0	R_0	ΔR_1	R_0^1	ΔR_2	R_0^2
(1, 2)	(1, 2)	(1, 5)	(4, 8)	(1, 7)	(4, 8)
(1, 3)	(1, 3)	(1, 6)	(5, 6)	(7, 4)	(5, 6)
(1, 4)	(1, 4)	(1, 8)	(6, 4)	(3, 7)	(6, 4)
(3, 4)	(2, 5)	(3, 8)	(6, 7)	(7, 7)	(6, 7)
(6, 4)	(2, 6)	(3, 6)	(6, 8)	(7, 8)	(6, 8)
(2, 5)	(3, 4)	(7, 6)	(7, 5)	(2, 8)	(7, 5)
(3, 5)	(3, 5)	(2, 4)		(5, 5)	
------	------				
(7, 5)	(4, 8)	(2, 7)		(5, 6)	
(2, 6)	(5, 6)	(2, 8)			
(5, 6)	(6, 4)	(5, 4)			
(6, 7)	(6, 7)	(5, 7)			
(4, 8)	(6, 8)	(5, 8)			
(6, 8)	(7, 5)	(6, 5)			

Figure 5.8: An Example of Computation of R_0^+ .

new tuples. These tuples, in the column of R_0 above the dotted line, can actually be removed from R_0 without affecting the final result. A new relation R_0^1 formed in this way can be used in the second iteration to compute ΔR_2. In this example, R_0^1 consists of only 6 tuples, less than 50 percent of R_0.

Figure 5.9 lists algorithm REDUCE, an algorithmic description of the suggested strategy for reducing the size of relation R_0. The notation used is similar to that used in the semi-naive algorithm: two relations to be joined in iteration i are ΔR_i and R_0^i. ΔR_i contains new tuples in the transitive closure generated in the $(i-1)^{th}$ iteration. Relation $R_0^0 = R_0$, and R_0^i is reduced to R_0^{i+1}, which is to be used in the next iteration to join with ΔR_{i+1}. Note that algorithm REDUCE as described above is for general cases. For a particular algorithm, for example, the semi-naive algorithm, the removal of tuples from ΔR_i is only needed for the first iteration of join R_0 and R_0: for the semi-naive algorithm, ΔR_i only contains newly generated tuples which are not in R_0.

Graphically, removing tuples as described in the algorithm is the process of removing outgoing edges from nodes satisfying the following conditions: (i) there is no incoming edge to the node, and (ii) all outgoing edges are already inserted to the relation. The second condition is automatically satisfied because the original relation R_0 is copied into the result. Since there is no incoming edge to the node, no more paths can be generated via the node, and its removal from the graph will not lose results. In the above example, node 1 has no incoming edges; after the edges started from it are inserted to the result relation, it can be removed along with those edges. This removal of node 1 further causes the removal of nodes 2 and 3, since only incoming edges for nodes 2 and

Algorithm REDUCE:
Input : Two intermediate relations ΔR_i and R_0^i
Output : Relation R_0^{i+1}

```
begin
  repeat
    foreach tuple t ∈ R₀ⁱ do
      begin
        if ΔRᵢ○t = ∅
        then begin
              remove t from R₀ⁱ;
              if t ∈ ΔRᵢ
              then remove t from ΔRᵢ;
            end;
      end;
    until no tuple can be removed from R₀ⁱ;
    R₀ⁱ⁺¹ := R₀ⁱ;
end;
```

Figure 5.9: Algorithm Reducing the Size of R0.

3 are from node 1.

For large database relations, it will be very expensive if algorithm REDUCE is implemented as it is described in figure 5.9. In the next section, one possible implementation is described which modifies the hash join method to dynamically reduce the size of R_0 without heavy overhead. Another point we would like to make is that this strategy has some flavor of using join indices to compute the transitive closure [Vald86]: only those tuples which are *joinable* are kept for computation. However, join indices are static data structures and do not change for different iterations of the computation. In our algorithm, size reduction is dynamically performed. We have the benefit of reducing the data size without the disadvantage associated with join indices: the costs of generating the join indices and maintaining them in a database; the difficulty of determining which relations and on which attributes the join indices should be maintained; and the complexity to determine whether it is beneficial to use the join indices.

5.6.2. Strategy 2: Speed Up the Convergence

The number of iterations needed to complete the transitive closure computation is another source of optimization. The logarithmic algorithm and smart algorithms outperform the semi-naive algorithm since they generate more tuples in one iteration and fewer iterations are needed. Intuitively, the source relations are only read from the disks once in one iteration. The more tuples generated in one iteration, the fewer

number of iterations needed to complete the computation. Thus, one of the major processing costs, disk I/Os for reading in the source relation, is reduced. The CPU cost, such as rehashing, if hash join is used, is also reduced partly. The savings gives the logarithmic and smart algorithm better performance [Vald86, Ioan86].

The recursive algorithm is an extreme along this direction: when a tuple is processed, all tuples derivable from this tuple are generated. The performance of the algorithm is irrelevant to the maximum path length of the transitive closure of the relation. If there are some very long paths in the transitive closure, this algorithm will outperform the iterative algorithms. The limitation of this algorithm is that, in order to find all tuples derivable from a tuple, the processing has the flavor of the depth-first search. In cases where the size of memory is much smaller than the relation size, a large amount of disk access is required, which leads to bad performance [Lu87].

The strategy suggested here combines the iterative methods with the recursive algorithm. For each pair of buckets which can be held in main memory, all tuples in the transitive closure derivable from them are generated. These tuples are output either to the corresponding buckets for further processing or to the final result relation.

Algorithm PROCESSING in figure 5.10 describes the algorithm of processing the ith bucket pair in the kth iteration using the strategy. ΔR_i^{k-1} contains the tuples generated in iteration $k-1$ and is hashed on the second attribute. $R_{0_i}^k$ is the corresponding bucket partitioned on the first attribute. R_i is the ith bucket of the result relation R, the transitive closure of R_0. Function GetBucketNo() returns the bucket number a tuple belongs to when hashing on the second attribute. The algorithm works as follows: for each tuple $t(a,b)$ in ΔR_i^{k-1}, it finds all matching tuples from $R_{0_i}^k$. New tuples are formed and hashed on the second attribute to find the buckets to which the tuples belong. The tuples falling to the current bucket are used to further probe the hash table. Tuples of other buckets are output to the corresponding buckets. They are either processed in the same iteration (if the bucket has not been processed yet), or processed in the next iteration. For each tuple, the processing will terminate when cyclic data (a tuple $t(a,a)$ is obtained) is encountered, or no more matching tuples can be found in $R_{0_i}^k$.

We use a simple example to explain the algorithm. Relation R_0 shown in figure 5.11 consists of 8 tuples. They are partitioned into two pairs of buckets, $(R_{0_1}^b, R_{0_1}^a)$ and $(R_{0_2}^b, R_{0_2}^a)$, on attribute b and a, respectively, because of the limitation of memory size. A tuple $t(a,b) \in R_{0_1}^b$ iff $hash(t.b)$ in $\{1, 2, 3\}$ and $t(a,b) \in R_{0_2}^b$ iff $hash(t.b)$ in $\{4, 5, 6\}$. Partitions $R_{0_1}^a$ and $R_{0_2}^a$ are formed in a similar way.

The computation starts with the first pair of buckets, $\Delta R_1^0 = R_{0_1}^b$ and $R_{0_1}^0 = R_{0_1}^a$. Algorithm PROCESSING is applied and the result tuples are hashed on the second attribute. Those tuples with hash values in $\{4, 5, 6\}$ (five of them in this example) are appended to the bucket ΔR_2^0 (as shown in the figure under the dotted line). Other tuples (in this case, three) are output as the result. The second pair of buckets is processed in a similar way. The difference is that the tuples generated with

Algorithm PROCESSING:

Input : A pair of buckets, ΔR_i^{k-1}, $R_{0_i}^{a}$

Output : Tuples in the transitive closure of R_0, which are inserted
into corresponding buckets

```
begin
    foreach tuple t in ΔR_i^k do
probe:
        if there is a match tuple t' in R_{0_i}^k with t.B = t'.A
        then begin
            form a new tuple newt(t.A, t'.B);
            j = GetBucketNo(t'.B);
            if (j > i)
            then output newt to ΔR_j^{k-1};
            if (j < i)
            then output newt to ΔR_j^k;
            if  (j = i)
            then if (t.A ≠ t'.B)
                then goto probe;
                else output newt into R_i;
        end;
end;
```

Figure 5.10: Algorithm PROCESSING.

the hash value of the second attribute in $\{1, 2, 3\}$ are used to form ΔR_1^1, which is used in the next iteration.[1]

This strategy can be explained intuitively with the graphic representation of R_0 as follows: The hashing technique partitions the directed graph, G_0 into a number of subgraphs G_{0_i}. An edge $e: a{\to}b$ is in subgraph G_{0_i} iff b is in bucket i. For each edge $e \in G_{0_i}$ ($a{\to}b$), algorithm PROCESSING finds all paths that start from node a and are contained in subgraph G_{0_i}. If there is a path leading to a node c in another subgraph, G_{0_j}, the output of a tuple (a,c) to bucket ΔR_j^b during the processing can be viewed as inserting a node a and an edge $a{\to}c$ in subgraph G_{0_j}. Therefore, any path starting from node a in subgraph G_{0_i} and ending with another node b in subgraph G_{0_j} can be

---

[1] In the example we did not show the elimination of duplicates: duplicates in the result tuples are eliminated before the next iteration, as when using the semi-naive algorithm.

Iteration 1			Iteration 2		
ΔR_i^0	$R_{0_i'}$	R_i^1	ΔR_i^1	$R_{0_i^2}$	R_i^2
(6, 1)	(1, 2)	(6, 2)	(3, 1)	(1, 2)	(3, 2)
(1, 2)	(1, 5)	(6, 3)	(4, 1)	(1, 5)	(3, 3)
(2, 3)	(2, 3)	(1, 3)	(5, 1)	(2, 3)	(4, 2)
(5, 3)	(3, 4)		(2, 1)	(3, 4)	(4, 3)
					(4, 4)
					(5, 2)
					(2, 2)
(3, 4)	(4, 6)	(3, 6)	------	(4, 6)	
(1, 5)	(5, 3)	(1, 6)	(3, 5)	(5, 3)	
(4, 6)	(5, 6)	(1, 1)	(4, 5)	(6, 1)	
(5, 6)	(6, 1)	(6, 6)	(5, 5)		
------		(2, 6)	(2, 5)		
(6, 4)					
(6, 5)					
(1, 4)					
(2, 4)					
(5, 4)					

Figure 5.11: An Example of Using Strategy 2.

internally found in subgraph G_{0_j} later on.

The effectiveness of this strategy is clearly shown by the example in figure 5.11. The longest path in the transitive closure includes five edges ($1\rightarrow2\rightarrow3\rightarrow4\rightarrow6\rightarrow1$), which requires five iterations for the semi-naive algorithms and three iterations for the logarithmic algorithm. However, only two iterations are needed using our strategy.

From the example, we can also see some savings other than the reduction of the number of iterations. In the previous iterative algorithms, new tuples generated during computation have to be read in at least once to join with the original relation. In our strategy, the result tuples corresponding to the paths which do not cross the border of subgraphs are not read in again. In the example, among 23 tuples generated in the transitive closure (excluding the original tuples in R_0), only 12 tuples are written out and then reread in for later processing.

5.6.3. Algorithm HYBRIDTC

In this section, we describe a hash-based transitive closure algorithm. It integrates the strategies described in the last section. Since this algorithm combines the merits of both iterative and recursive methods, we name it algorithm HYBRIDTC (a *hybrid* transitive closure algorithm).

5.6.3.1. The Algorithm

> **Algorithm HYBRIDTC;**
> **Input** : relation R_0
> **Output:** relation R, the transitive closure of relation R_0
>
> **begin**
>   partition R_0 on $R_0.A$ and $R_0.B$ into
>   buckets $R_{0_i}{}^a$ and $R_{0_i}{}^b$ $(1 \le i \le N)$;
>   **for** i := 1 **to** N **do begin**
> $\Delta R_i{}^1 := R_{0_i}{}^b$;
> $R_{0_i}{}^1 := R_{0_i}{}^a$;
>   **end;**
>   k := 0;
>   **repeat**
>   k := k + 1;
>   **for** i := 1 **to** N **do**
>   **if** $(\Delta R_i{}^k \ne \varnothing)$ and $(R_{0_i}{}^k \ne \varnothing)$
>   **then** ProcessingBucket (i, k, ΔR_i^k, $R_{0_i}{}^k$);
>   **else** $\Delta R_i{}^k := \varnothing$;
>   **for** i := 1 **to** N **do begin**
> $\Delta R_i{}^k := \Delta R_i{}^k - R_i{}^{k-1}$;
>   **until** all $\Delta R_i{}^k$s are empty;
> $R := \bigcup_{1 \le i \le N} R_i$;
> **end.**

Figure 5.12: Algorithm HYBRIDTC.

The algorithm is shown in figure 5.12. Relation R_0 is partitioned into two sets of buckets on attribute $R_0.A$ and $R_0.B$ as in traditional hash joins. These two set of buckets are denoted by $R_{0_i}{}^b$ and $R_{0_i}{}^a$ $(1 \le i \le N)$, respectively. We will use subscripts to denote the bucket number and superscripts to denote the iteration number. Let $\Delta R_i{}^k$ contain the new tuples in the transitive closure that belong to bucket i (hashed on attribute B) generated during the $(k-1)^{th}$ iteration, and $R_{0_i}{}^k$ be the reduced bucket i of R_0 after $(k-1)$ iterations. The bucket pair processed in the kth iteration is $\Delta R_i{}^k$ and $R_{0_i}{}^k$, where $\Delta R_i{}^1 = R_{0_i}{}^b$, and $R_{0_i}{}^1 = R_{0_i}{}^a$.

After the relation is partitioned, the ΔRs are initialized to be the corresponding set of buckets. The processing of bucket pairs proceeds iteratively until all $\Delta R_i{}^k$s are empty for the kth iteration. Since $\Delta R_i{}^k$ contains the most recently generated tuples, and $R_{0_i}{}^k$ is also reduced during each iteration, procedure *ProcessingBucket* is only

called when both of them are nonempty. During the processing of bucket pair ΔR_i^k and $R_{0_i}^k$, some result tuples are inserted into R_i, and others are inserted to other buckets ΔR_j, $(j \neq i)$, as described in algorithm PROCESSING. After each iteration k, duplicates are eliminated from the ΔR_i^{k+1}'s which are going to be used in the next iteration.

```
procedure ProcessingBucket ( bucketno, iteration : integer;
                 deltabucket, bucketR0 : buckets );
begin
  BuildHashTable(bucketR0);
  foreach tuple in deltabucket do
    ProcessingTuple(bucketno, iteration, tuple);
  foreach tuple in the hash table do
    if   tuple.mark
    then   OutputBucketR0 (tuple, bucketno, iteration+1);
end;
```

Figure 5.13: Procedures *ProcessingBucket*.

The union and duplicate elimination procedures are the same as any transitive closure algorithms, and we are not going to discuss them here. Figure 5.13 and Figure 5.14 give one possible implementation of the procedures *ProcessingBucket* and *ProcessingTuple*. In this implementation, a hash table is constructed for R_{0_i} as in the traditional hash join algorithms. However, one extra field "*mark*" is added to the hash table entry. It is used to mark the tuples actually participating in the join. Procedure *ProcessingTuple* is called for each tuple in *deltabucket* (ΔR_i^k). After all tuples have been processed, only those marked tuples are written back by calling procedure OutputBucketR0 to form $R_{0_i}^{k-1}$.

Procedure *ProcessingTuple* implements strategy 2 using a stack of tuples. *PushStack, PopStack*, and *EmptyStack* are procedures and functions manipulating the stack. The tuple on the top of the stack is used to look up the hash table to find matches. Those matching tuples can be divided into three categories according to the bucket it belongs to. The bucket number of a tuple is returned by function GetBucketNo. The tuples of other buckets are inserted to ΔR_j buckets by procedure OutputDelta. The tuples of current processing buckets are pushed onto the stack for later processing. This process continues until the stack is empty.

The advantage of using a stack is its simplicity. Another advantage, perhaps a more important one, is ease of memory management. If there is a large number of tuples derived from some particular tuple in the bucket which leads to a full stack, we can just write part of the bottom of the stack on the disk and reread it back in to free memory space later on for continuing the process. Thus, algorithm HYBRIDTC does not introduce new issues in memory management. Techniques of partitioning a relation

```
procedure ProcessingTuple( bucketno, iteration: integer;
                    inputuple : TupleType );
var  currenttuple, matchtuple, newtuple : TupleType;
     newbucketno : integer;
begin
  PushStack(inputuple);
  while (NOT EmptyStack) do
    begin
      currenttuple := PopStack;
      if (currenttuple.a <> currenttuple.b)
      then begin
           matchtuple := LookUp(currenttuple);
           foreach  matchtuple do
             begin
               if (NOT matchtuple.mark)
               then matchtuple.mark := true;
               newtuple := FormTuple (currenttuple.a, matchtuple.b);
               newbucketno := GetBucketNo(newtuple);
               if (newbucketno = bucketno)
               then PushStack(newtuple);
               if (newbucketno < bucketno)
               then OutputDelta(newtuple, newbucketno, iteration+1);
               if (newbucketno > bucketno)
               then OutputDelta(newtuple, newbucketno, iteration);
             end;
         end;
      OutputResult(bucketno, currenttuple);
    end;
end;  (* procedure ProcessingTuple *)
```

Figure 5.14: Procedure of Processing a Tuple in ΔR_i^k.

into buckets and of handling overflow buckets developed in hash join methods can be directly used.

Now, we prove the following Lemma:

Lemma: *Algorithm HYBRIDTC correctly computes the transitive closure of a database relation.*

Proof: The proof of the Lemma consists of two parts. First, we have already explained in Section 5.6.2 that the removal of unmarked tuples, the tuples not participating in the join in the current iteration, will not lead to loss of the result tuples. Second, we prove that the algorithm will find all tuples in the transitive closure. In other words, the

algorithm can find all paths in graph G_0 if relation R_0 is represented by G_0. Let p be a path of graph G_0. It is obvious that, if all nodes on path p are contained in one subgraph of G_0, the path can be found by the algorithm when the corresponding buckets are processed. It is more likely that paths cross over the border of subgraphs. Let e, $(a{\rightarrow}b) \in G_0$, be an edge, and the end nodes of e be a and b, and they are in two different subgraphs, G_{0_i} and G_{0_j}, respectively. Then tuple (a,b) is in bucket j. During processing of bucket j, all paths of p starting from b and ending at some nodes y_j in G_{0_j} can be found, and a set of tuples $\{ (b, y_1), \ldots , (b, y_j), \ldots\}$ is generated. If there are some paths starting from some node x_i and ending at node a, the processing of bucket i will not only generate a set of tuples $\{x_i,a\}$, but also generate a set of tuples $\{x_i,b\}$. They are inserted into bucket j. Thus, in the next iteration of processing bucket j, all paths starting from node x_i and ending at node y_j can be found. The proof can be extended to the paths across any number of subgraphs. $\quad\square$

5.6.3.2. Performance Comparisons

Qualitatively, algorithm HYBRIDTC is expected to improve bad performance in the following ways:

(1) *Reduce the number of iterations.*

For the semi-naive and logarithmic algorithms, only paths with certain lengths can be found in each iteration. The number of iterations needed to complete the computation is determined by the depth of the transitive closure, that is, the longest path. For algorithm HYBRIDTC, paths contained in a subgraph can be generated in a single iteration no matter how long it is. Furthermore, the later processed buckets make use of the new tuples generated by the buckets which have been processed in the same iteration. As a result, the number of iterations needed largely depends on how the relation is partitioned and is usually less than the depth of the transitive closure. The reduction in the number of iterations at least reduces the disk I/O needed to read in R_0 and CPU time for constructing the hash tables.

(2) *Reduce the number of disk I/Os needed to read in the delta relations.*

For both the semi-naive and logarithmic algorithms, the result tuples generated in one iteration have to be written to the disk and read in again in the next iteration. However, in algorithm HYBRIDTC, the tuples generated in one iteration need to be read in again only if they belong to other buckets. Again, the extent of this savings largely depends on the data distribution and the partitions.

(3) *Reduce the size of the source relation.*

The source relation used to compute the transitive closure is dynamically reduced during processing, compared to the constant size in the semi-naive algorithm and no optimization in the logarithmic algorithm.

Any quantitative analysis of algorithm HYBRIDTC is difficult, since the performance will vary dramatically with different data characteristics and the partitioning. In

order to validate our qualitative analysis above, we made some comparisons between the performance of the semi-naive algorithm, the logarithmic algorithm, and algorithm HYBRIDTC as follows:

(1) The data model proposed by Bancilhon and Ramakrishnan [Banc86] is used. We examined two simple cases, lists and trees having fanout 2.

(2) We use the number of tuples read in during the computation as the performance measure for the comparison. This number roughly reflects the total costs of the computation. The larger the number is, the more disk I/O cost and CPU cost for constructing the hash tables. Furthermore, we assume that duplication elimination costs are the same for all three algorithms, and they are not taken into account.

(3) Some of the implementation details are ignored. For example, for the semi-naive algorithm and the logarithmic algorithm, we only calculate the total number of tuples of two relations joined in each relation. This number is therefore independent of the memory size and the number of hash buckets. We actually assume that the pipeline method is used to reduce the number of disk I/Os [Lu87]. That is, each tuple in the transitive closure only counts once: no separate partition phase is assumed.

With the above assumptions, the total number of tuples for the semi-naive and the logarithmic algorithms are calculated as follows:

For the semi-naive algorithm, h iterations are needed to generate all tuples in the transitive closure. One more iteration is actually completed, resulting in the termination of the computation. During each iteration, there is only one join. The total number of tuples participating in the join operations is:

$$N_{semi-naive} = \|R\| + (h+1)\|R_0\|$$

The number of iterations needed in the logarithmic algorithm, k, is determined by $k = \lg(h+1) - 1$. For each iteration i, there are two joins: the join of R_{2^i-1} with R^i, and the join of the result tuples in the transitive closure so far, which is $\sum_{j=1} R^j$, with the newly generated relation R^{2^i}. The total number of tuples participating in the computation is:

$$N_{logarithmic} = \sum_{i=1}^{k} (2^*R_0^i + \sum_{j=0}^{2^i} R_0^j)$$

The number of tuples read in algorithm HYBRIDTC is obtained by simulation: a program was coded to implement the algorithm in memory. A random number generator was used to assign bucket numbers for tuples. The corresponding buckets were then joined iteratively to compute the transitive closure. When each bucket pair was processed, the number of tuples in the buckets was counted. The total number of tuples read in could thus be obtained. In the simulation, we used a small bucket size (typically each bucket contains 10 tuples). Therefore the simulation actually does not favor algorithm HYBRIDTC.
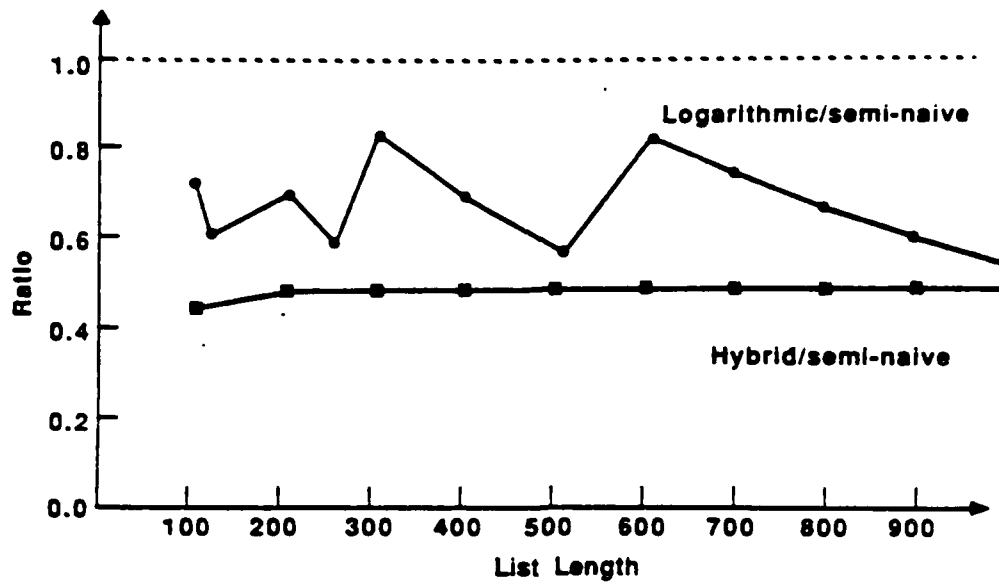
Figure 5.15: The Performance Comparison 1 (R_0: List).
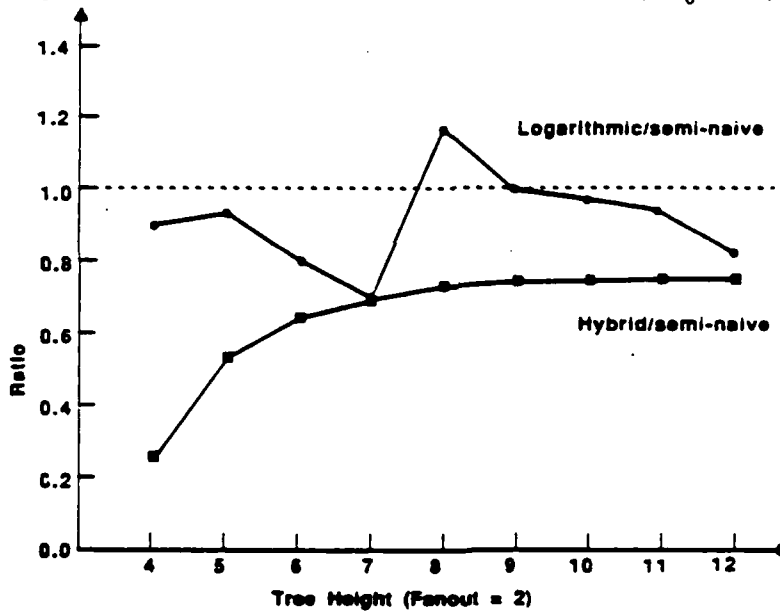
Figure 5.16: The Performance Comparison 2 (R_0: Tree).

The result of this comparison is shown in figures 5.15 and 5.16. The lengths of the lists vary from 100 to 1024. The tree depth varies from 4 to 12. The comparison uses the number of tuples in the semi-naive algorithm as a reference. The ratio of logarithmic/semi-naive and hybrid/semi-naive are computed. The results in the figures show that algorithm HYBRIDTC consistently outperforms the other two algorithms. For lists, the ratio hybrid/semi-naive is about 50 percent. However, the ratio of logarithmic/semi-naive is about 60 to 70 percent. This result is expected as we discussed above.

In both figures, the ratio of logarithmic to semi-naive is not monotonic. Sometimes, the semi-naive algorithm even outperform the logarithmic algorithm. This happens when the depth is just larger than 2^k. This is also observed by Ioannidis [Ioan86]. The explanation is that the number of iterations of the logarithmic algorithm is determined by the depth. When the depth increases to past 2^k, the number of iterations increases by 1. That is, another iteration is required to complete the computation to find just a few more tuples. That is one disadvantage of the logarithmic algorithm.
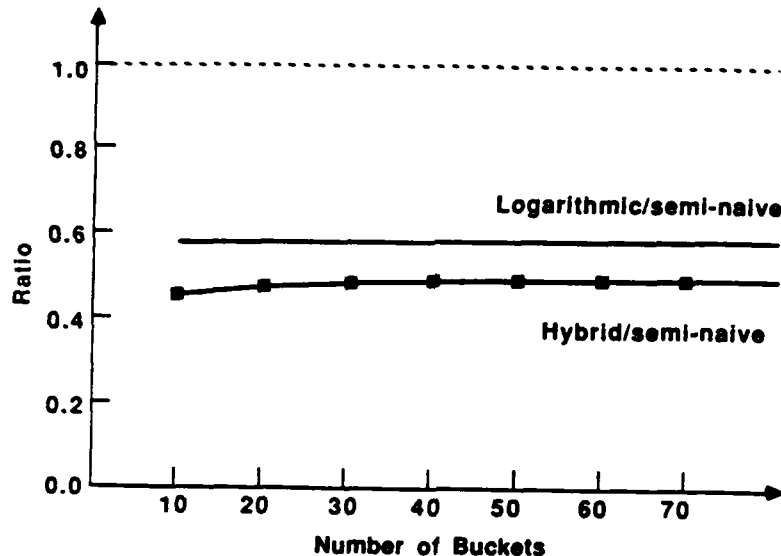
Figure 5.17: The Performance versus Number of Buckets (R_0: List).

We did not compare the performance of algorithm HYBRIDTC with the recursive algorithm. Its performance becomes much worse than the logarithmic algorithm when the memory size is small, compared with the relation size [Lu87]. However, algorithm HYBRIDTC still performs better than the other two algorithms, even in this case. Figure 5.17 illustrates the number of disk I/O tuples with the different number of buckets into which the relation is partitioned. When we increase the number of buckets, which simulates smaller and smaller bucket size, the number of disk I/O tuples also increases. However, it is still less than what needed in the other two algorithms.

5.6.4. Conclusions

We have discussed two strategies which optimize the computation of the transitive closure of a database relation. We also presented a hash-based algorithm that integrates these two strategies together. The algorithm is easy to implement in real systems by modifying the traditional hash join methods. A simple performance analysis was conducted, and the results indicate that the new algorithm does outperform previous algorithms. This performance analysis is far from complete. However, it does provide the evidence that our new strategies in optimization are in the right direction. Further detailed implementation in relational database systems and performance analysis is one of the possible projects for future work.

Besides better performance, the algorithm has some other advantages. For example, the algorithm is easy to extend to become a distributed or parallel algorithm. In algorithm HYBRIDTC, there is no inherent sequence among the iterations. For other algorithms, the result of an iteration is used as the input of the next iteration. In the logarithmic algorithm the second join in each iteration can only be started after the first join finishes. For the distributed version of algorithm HYBRIDTC, each processor or node can work on one or more pairs of buckets. The tuples generated at one processor are either processed locally or sent to other processors. The only synchronization needed is the final termination of the whole computation.

This algorithm can be further optimized along the directions proposed. One possibility is as follows: the new tuples generated are not only hashed on the second attribute and inserted into the corresponding buckets, but also hashed on the first attribute and inserted into the second relation in the join ($R_0{}^k$). Thus, more tuples can be generated in each iteration, and performance improvement can be expected. However, it is somewhat difficult to implement in real system since the size of $R_0{}^k$ will change during processing. Some sophisticated memory management strategy and bucket overflow techniques have to be developed.

Algorithm HYBRIDTC is a basic algorithm for computing the simple transitive closure of a relational database. Interesting future work is to use it as a base for extending a relational database management system to include transitive closure as one basic operation. To achieve this, the algorithm should be further augmented so that more complicated transitive closure queries can be processed efficiently [Agra87].

CHAPTER 6

Parallel Architectures for Database Management

As enterprises use database management systems to manage more of their information, the size of existing databases is increasing rapidly. Databases of over 100 gigabytes now exist; terabyte databases, if they do not exist now, will appear in the next few years. Managing these large databases will require more powerful architectures than are in common use today. Technology now permits the construction of multiprocessor database management architectures with tens or hundreds of processors, a gigabyte or more of main memory, and disk capacity in the terabyte range. The Teradata DBC/1012 is an example of such an architecture [83].

The join operation is an important operation for relational database systems, and will become even more important as logic-based inference capabilities are added to these systems. In this chapter, we describe a number of multiprocessor join algorithms. The algorithms use sort-merge and hashing techniques, and are highly parallel and pipelined. The algorithms are designed to execute on a multiprocessor architecture that is parameterized in the degree of memory sharing, so that tightly coupled, loosely coupled, and intermediate architectures can be modeled. Other architectural parameters include the number of processors, number of disks, amount of main memory, and interconnection network bandwidth. We model the performance of the algorithms analytically to determine elapsed time, resource utilization, and other quantities as functions of the workload and architectural parameters. The join algorithms overlap computation, disk transfers, and interconnection network transfers. The analysis models this overlap and identifies bottlenecks that limit the algorithms' performance. We do not model multiple simultaneous join operations, and therefore do not compute system throughput. Based on this analysis, we answer the following questions:

- How do the algorithms compare in performance? When does one outperform another?
- How does response time vary as a function of the architectural parameters?
- How does response time vary with the workload?
- Does shared memory help algorithm performance? To what extent?
- What are the architectural bottlenecks? How could they be alleviated?

In the following sections, we describe the multiprocessor hardware architecture and the join algorithms, develop cost formulas for the algorithms, compare the algorithms' performance under for various workloads and hardware configurations, and summarize the results of our investigation.

6.1. Multiprocessor Data Management Architecture

Many specialized architectures have been proposed for high performance relational database management. These architectures include logic-on-disk machines [Schu79, Su79], VLSI-based special purpose processors [Kits83, Shib84], and loosely- and tightly-coupled multiprocessor architectures [DeWi79, 83, DeWi86]. We believe that commercially viable database machines must be constructed principally from commodity components such as general purpose microprocessors and conventional disk storage devices. This belief is based on the superior price/performance and reliability of commodity components compared to custom components. Therefore, we consider in this study a multiprocessor architecture with the following characteristics:

- The architecture uses a large number (tens to hundreds, at least) of processors to obtain the necessary performance. This assumes that the processors can be used effectively. The Teradata DBC/1012 appears to have demonstrated that this is possible.

- The architecture can use large amounts (hundreds of megabytes to hundreds of gigabytes) of semiconductor memory. In the next few years, this amount of memory will be feasible as well as cost-effective.

- The architecture can support an aggregate disk capacity of a terabyte or more; only a small fraction of the total database can be accommodated in main memory. We assume further that many of the individual database relations will typically not fit in main memory.

Figure 6.1 shows a block diagram of our architecture. The architecture consists of a set of *clusters* linked by an intercluster bus or ring. Each cluster consists of a set of processors, a shared memory bank addressable by all the processors in the cluster, and a set of disk storage units and associated controllers. Processors read and write the shared memory in units of a few bytes, with little contention. The processors may have local caches to reduce memory contention, but this is invisible to the data management software except for possibly the need to flush the cache occasionally. Transfers between disk and memory, and between cluster memories over the bus, are a page at a time, where a page is a few kilobytes or more in size. A specific configuration of this architecture is determined by the following parameters:

NC number of clusters
NP number of processors *per cluster*
ND number of disks *per cluster*
M pages of main memory available *per cluster*
PG page size in bytes

These parameters can be varied to determine the effect of architectural changes. For instance, if the CPU is a bottleneck, more CPUs can be added per cluster or each CPU can be made faster. (CPU speed is defined in terms of execution times for basic operations associated with the join algorithms, such as tuple move.) If the disk is a bottleneck, the page size can be increased or more disks can be added per cluster. We have assumed that the network is a single bus, so the only architectural cure for a
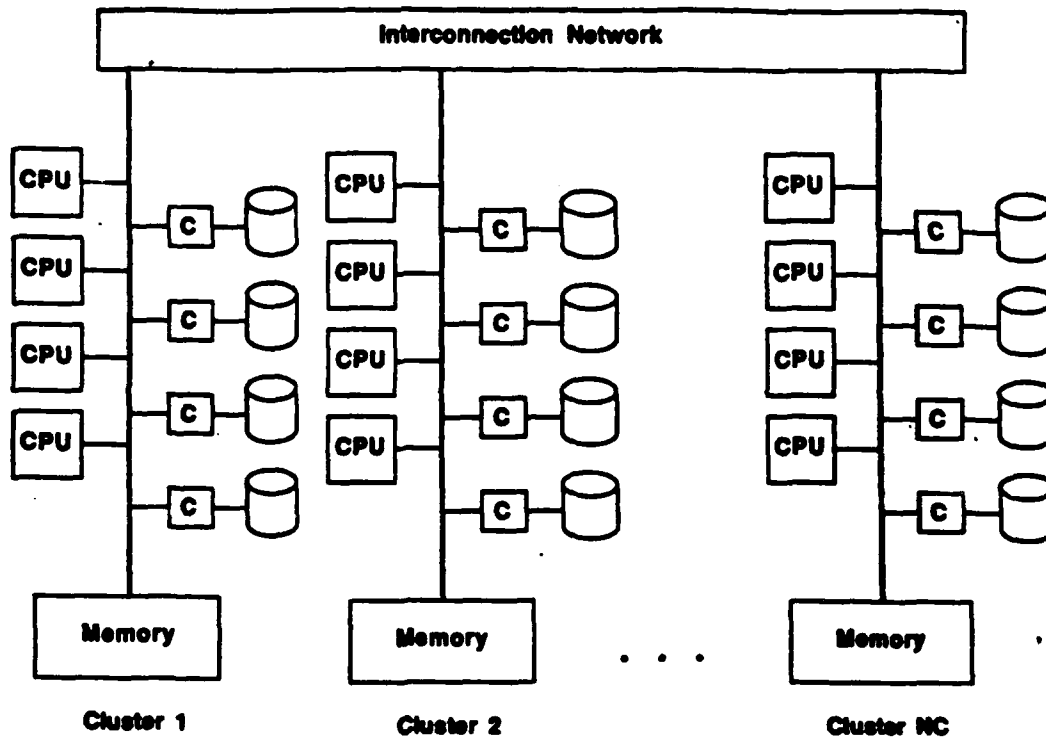
Figure 6.1. Multiprocessor Data Management Architecture

network bottleneck is to increase the network transmission rate.

6.2. Join Algorithm Descriptions

The problem that each join algorithm solves is the following: given relations R and S, compute their natural join on an unspecified pair of attributes, giving output relation O. R and S are assumed to be uniformly partitioned across all disks on all clusters. The partition is not determined by the values of the join attributes, so that tuples must be transmitted between clusters to perform the join. Let R_i and S_i denote the fragments of R and S, respectively, stored on the disks at cluster C_i. The result of the join can be partitioned across the clusters; it need not be collected on one cluster. No projection is performed on the result except to remove the redundant copy of the join attribute, so no duplicate tuples are produced.

Assume without loss of generality that S is larger (in bytes) than R. Some of the algorithms transmit only one of the relations over the interconnection network; they transmit R, the smaller relation.

Each algorithm consists of one or more *phases*. The phases are executed one after another; the activity in one phase completes before the next phase starts. Within each

phase, a fixed set of processes execute in parallel, passing data to each other (possibly over the network) and reading from and writing to disk in pipelined fashion. Each cluster has dedicated send and receive processes to act as intermediaries between communicating processes on different clusters.

Processes communicate with each other via streams of data pages. Each process may have several input and output streams. We chose this granularity of communication to minimize the interprocess communication and synchronization overhead. The cost of passing a page of data between processes on the same cluster is assumed to be negligible in comparison to the cost of producing or consuming it. In some algorithms, processes on the same cluster read concurrently from the same page buffer or other memory area; concurrent reading and writing is not used because it would require a high synchronization overhead. The interprocess communication system uses flow control to match the speeds of the producing and consuming processes and to prevent buffer overflow. Enough buffers are allocated to allow all processes to execute concurrently.

We present six algorithms here. The algorithms come in pairs: the first of each pair transmits tuples from both R and S over the network, while the second transmits only tuples from R, reducing communications cost while increasing computation. The two algorithms comprising the first pair are parallel versions of the the basic sort merge join; they are most similar to algorithms described in [Bitt83, Vald84]. The other four use hashing to decompose R and S into buckets, and then use either a sort-merge or a hashing technique to join each pair of buckets. They are similar to algorithms described in [Kits83, DeWi85].

6.2.1. Parallel Sort Merge Algorithms

6.2.1.1. Parallel Sort-Merge Join Type 1 (SMJ1)

In the basic sort-merge join, each relation is first sorted on its join attribute. Then, the two sorted relations are merge-joined. The merge-join operation matches tuples in the two relations by their join attributes, and generates the result tuples. It is pipelined much like a merge operation, except that a tuple in either input relation can be used to construct multiple output tuples.

Previously published algorithms have presented parallel algorithms for the sort phase. Bitton *et al.* describe join algorithms employing a parallel binary merge sort and a block bitonic sort [Bitt83]. The former can be improved, memory permitting, by using a general multi-way merge [Vald84]. Both algorithms start by generating a set of sorted runs from the original unsorted relation. These runs are generated using a main-memory sorting algorithm or a priority queue. The latter is preferable because it generates runs that are on average twice the size of the main memory dedicated to the priority queue, and hence twice the size of the runs generated by a main-memory sorting algorithm [Knut73]. In addition, run generation by priority queue is inherently a pipelined operation, permitting better overlap between CPU and I/O than run generation by main memory sorting. Once the runs are generated, the final sorted output is

generated either by merging the runs or by using a block bitonic algorithm.

The sort merge join algorithm just described parallelizes the sort operations. but the final merge-join operation is still performed sequentially over the entire length of both relations. In the algorithm described below, the final merge-join is partitioned into multiple parallel processes so that no single process must pass over all of either relation. This is accomplished by generating $NFRUN_R$ final runs of relation R and $NFRUN_S$ final runs of relation S, and merge-joining each of the final runs of R with each of the final runs of S, all in parallel. One of these merge-joins is executed at each cluster, leaving the joined relation O partitioned across the clusters. There is an obvious constraint

$$NFRUN_R \cdot NFRUN_S = NC$$

For convenience in describing the algorithm, let the clusters be named C_{ij} for $1 \le i \le NFRUN_R$ and $i \le j \le NFRUN_S$. The clusters are logically arranged as a two-dimensional array with $NFRUN_R$ rows and $NFRUN_S$ columns, though their physical interconnection is unchanged. (Assume for now that there are at least two rows and two columns. The degenerate case is discussed below.) The portion of R stored on C_{ij} is called R_{ij}.

The algorithm has three major phases.

Phase 1: At each cluster C_{ij}, NP processes generate initial runs of R_{ij} and write them back to disk. (NP is the number of processors per cluster.) Each process has its own priority queue for generating the runs.

Phase 2: Each cluster C_{ij} generates initial runs of S_{ij} in the same way.

Phase 3: Merge the initial runs of R into $NFRUN_R$ final runs, and the initial runs of S into $NFRUN_S$ final runs. Also, merge-join each final run of R with each final run of S to produce the result. The merging and merge-joining form a five-stage pipeline. The final runs of R are produced in two stages. The first stage of the merge occurs at each cluster C_{ij}, where a merge process merges all initial runs of R_{ij} into a single sorted version of R_{ij}. One of the clusters in each row, the *row pivot cluster*, executes the second stage, merging the sorted R_{ij}'s into a final sorted run for the row. The final runs of S are generated in the same way except that a *column pivot cluster* executes the second merge stage for clusters in its column. The row (column) pivot clusters send their final runs to the other clusters in the row (column); each cluster merge-joins the final run of R for row i with the final run of S for column j to produce O_{ij}, a fragment of the final result. By choosing C_{ii} as the row pivot cluster for row i, and $C_{j \oplus 1 \; j}$ as the column pivot cluster of column j, no cluster is both a row and a column pivot.

6.2.1.2. Parallel Sort-Merge Join Type 2 (SMJ2)

It may be wasteful to sort both relations completely before merge-joining them, especially if the join selectivity is low. At each stage of the merging process, tuples are being processed that may not participate in the final join. The following algorithm makes only one pass over the larger relation (S). In the description, we revert to single

subscripts on clusters and relation fragments.

The algorithm has three phases.

Phase 1: Generate local runs of R_i at each cluster C_i, as in algorithm SMJ1.

Phase 2: Merge these runs into a single sorted version of R. This is done with a two-stage merge. Each cluster executes the first stage, merging the local runs into a sorted version of R_i. The results from each cluster are sent to C_1, which executes the second stage of the merge and broadcasts the resulting sorted R to all clusters. Each cluster writes this run to disk.

Phase 3: Generate runs of S at each cluster using a priority queue. However, instead of writing these runs to disk, merge-join them immediately with R to produce the output tuples. Let there be NP processes at each cluster executing the run generation stage, paired with an equal number of processes executing the merge-join stage.

The advantage of this algorithm is that it produces the join results with one pass over the S relation. When S is large, this presumably saves much of the I/O and processing that would otherwise be required to produce the final runs of S. The disadvantage is that R must be read repeatedly from disk to be joined against the runs of S. (If R fits entirely in main memory, this is unnecessary. However, hash-based algorithms may be superior in this case. We do not attempt to fit all of R in main memory.)

6.2.2. Hash Partitioning Join Algorithms

These algorithms all use a hash partitioning technique described in [DeWi84] to decompose a join of two large relations into a sequence of smaller joins. They partition tuples of R and S into *batches* $RB_0, \ldots, RB_{NBATCH}$ and $SB_0, \ldots, SB_{NBATCH}$ and join the respective batches. The partitioning is based on the value of a hash function applied to the join attribute, so that joining the respective batches generates all required result tuples. The batches are sized so that each batch join can be performed in main memory. Batches $1, \ldots, NBATCH$ of each relation are written to disk during partitioning. Batches RB_0 and SB_0 are joined during or immediately after partitioning, depending on the particular algorithm. If sufficient memory is available, $NBATCH = 0$ and no batches must be written to disk. Otherwise, batches $1, \ldots, NBATCH$ are read from disk and joined one after the other. The number of batches can be computed from the size of the relations and available memory; see Section 6.3. This partitioning technique minimizes the amount of intermediate data that must be written to disk.

All of the algorithms partition the batches further into *buckets* and join the buckets within each batch in parallel. Two of the algorithms, HSM1 and HSM2, join respective buckets using a sort-merge technique similar to the GRACE algorithm [Kits83]. The other two, HH1 and HH2, use the in-memory hash table technique described in [DeWi84, Brat84]. The type 1 algorithms HSM1 and HSM2 transmit both R and S over the network; the type 2 algorithms HSM2 and HH2 transmit only R, the smaller relation.

6.2.2.1. Hash Based Sort Merge Join Type 1 (HSM1)

This algorithm partitions each batch of R and S further into $NC \cdot NP_{join}$ buckets that are joined in parallel by NP_{join} processes on each of the NC clusters. A hash function applied to the join attribute of each tuple determines the batch, cluster, and process in which it will be joined. Each pair of buckets is joined using a sort-merge join.

Let RB_{ij} denote the subset of RB_j derived from R_i, the subset of R stored at C_i. Define SB_{ij} similarly.

The algorithm has three phases.

Phase 1: At each cluster C_i, NP_{join} processes read R_i from disk a page at a time. The assignment of pages to processes is arbitrary. The processes hashes each tuple in a page to determine the batch, cluster, and process in which it will be joined. If the tuple is in batch $j \neq 0$, it is placed in a buffer to be written to a disk file for RB_{ij}. If the tuple belongs to batch 0 but is to be joined on a different cluster, it is placed in a buffer to be sent to that cluster. If the tuple is to be joined by a different process on the same cluster, it is placed in a buffer to be sent to the correct process. When a process fills one of these buffers, it writes it to disk or sends it to another cluster or process as appropriate. When a page is sent to another cluster, it is received by an arbitrary process on that cluster; the tuples in the page are rehashed and sent to another process in the cluster if necessary. Once the tuple arrives at the correct process, it is inserted into a binary search tree. The search tree will be traversed inorder in the next phase to produce a sorted version of the bucket.

When all of R_i has been read, S_i is processed in the same way.

Phase 2: This phase is repeated for j ranging from 1 to $NBATCH$. If $NBATCH = 0$, this phase is omitted.

At each cluster C_i, each process performs an inorder traversal of its R and S search trees to produce a sorted stream of tuples for each bucket. It merge-joins these tuples to produce the join results. As tuples are consumed, the space they occupied is freed. As the memory is freed, the processes read the file for batch RB_{ij}, hash each tuple, send the tuples to the appropriate cluster and process, and insert them into search trees occupying the newly freed space. When all of RB_{ij} has been read, SB_{ij} is processed in the same way.

Phase 3: In this phase, the buckets in batch $NBATCH$ are joined as described for phase 2. No data remains to be read from disk and partitioned.

6.2.2.2. Hash Based Sort-Merge Join Type 2 (HSM2)

This algorithm differs from algorithm HSM1 in that relation S is not sent over the network. Instead, each cluster C_i joins all of R with its portion S_i of relation S. A hash function on the join attribute of each tuple determines the batch to which the tuple belongs, and the process on each cluster (in the case of R) or the process on cluster C_i (in the case of S_i) that will do the joining.

The algorithm has three phases. Only the first phase will be described; the other phases should be clear from the description of phase 1 and Algorithm HSM1.

Phase 1: At each cluster C_i, NP_{hash} processes read R_i. They hash each tuple to determine the batch and process number. Each tuple is buffered either to be written to disk or to be broadcast to the correct process on each cluster. Each process constructs a binary search tree of tuples belonging to its own bucket. When all of R_i has been read, S_i is processed in the same way, except that tuples from S_i are not transmitted to other clusters, only to other processes on the same cluster.

6.2.2.3. Multiprocessor Hybrid Hash Join Type 1 (HH1)

Algorithm HH1 partitions R and S into batches and buckets in the same ways as Algorithm HSM1. However, it uses a hash-based algorithm to join each pair of buckets. An in-memory hash table is constructed for each bucket of R. This table is then probed using tuples from the corresponding bucket of S to produce the join results. The concurrent processing of consecutive batches performed in phase 2 of Algorithm HSM1 is not possible here because the hash table for a bucket of R cannot be deallocated until it has been probed by all the tuples in the corresponding S bucket. On the other hand, Algorithm HSM1 requires memory to hold S buckets, while this algorithm does not.

The algorithm has four phases.

Phase 1: At each cluster C_i, NP_{join} processes read R_i from disk. They hash each tuple and copy it to the appropriate buffer if it must be written back to disk or sent to another cluster or process, as in Algorithm HSM1. Each process constructs a hash table of tuples belonging to its own bucket.

Phase 2: At each cluster C_i, the NP_{join} processes read S_i from disk. They hash each tuple and buffer it to be written to disk or sent to another cluster if necessary. It is not necessary to send a tuple from one process to another on the same cluster, however. Once a tuple is at the correct cluster, any process can probe the appropriate hash table to generate the join results.

Phases 3 and 4 are repeated for j ranging from 1 to $NBATCH$. If $NBATCH=0$, these phases are omitted.

Phase 3: This is similar to phase 1 except that each cluster C_i reads RB_{ij} from disk instead of R_i, and performs no disk writes.

Phase 4: This is similar to phase 2 except that each cluster C_i reads SB_{ij} from disk instead of S_i, and performs no disk writes.

6.2.2.4. Multiprocessor Hybrid Hash Join Type 2 (HH2)

This algorithm is to Algorithm HH1 as Algorithm HSM2 is to Algorithm HSM1. It has four phases similar to those of Algorithm HH1. However, tuples of S are not transmitted over the network. In fact, they are not even transmitted between processes on the same cluster since any process can probe a hash table on the same cluster.

6.2.3. Discussion

The algorithms described above represent the latest versions in a sequence of algorithms. These versions provide better overlap in the usage of different resources than earlier versions. For example, in all four hash partitioning algorithms, the communications load is spread as evenly as possible over the duration of the algorithm execution. Tuples are sent across the network only when they are about to participate in a join. One of our earlier versions transmitted all tuples to the joining cluster when the relations were being partitioned, as in DeWitt and Gerber's algorithm [DeWi85]. We found that this could cause a network bottleneck during partitioning: the disks and CPUs were not well utilized. Spreading the communications load over the duration of the algorithms reduced their execution time.

The overlapping among disk I/O, CPU processing and data transfer over the network gives algorithm designers opportunities to tune the algorithms to obtain the desired tradeoff between the elapsed time, total processing cost and memory usage which is best for their system. Another example regarding this is the way the hash-based algorithms store tuples in batches $1 - NBATCH$. One possibility is to use one file per batch at each cluster; we chose to use one file per remote cluster at each cluster. In the former case, no repartitioning is needed in the later phases, but more buffer pages have to be allocated in the partitioning phase. In the latter case, the tuples in the same batch have to be rehashed to determine its bucket number, but far fewer buffer pages are needed to hold the tuples rewritten to disks.

6.3. Performance Comparisons

This section presents performance comparisons of the six join algorithms described in Section 6.2. These comparisons are based on the formulas obtained from the analysis.

The main purpose of the performance comparison is to get some insight of the behavior of different algorithms. The novelty of this performance analysis lies in two facts. First, there are few comprehensive performance studies of join algorithms in the multiprocessor-multi disk environment [DeWi85]. Second, most performance studies use total processing time as the metric. In fact, disk I/O operations, data transfer along the network and CPU processing are often overlapped. The extent of the overlap varies among different algorithms which leads to different elapsed times even with the same total processing cost. One of the goals of parallel join algorithm design should be to overlap the usage of various system resources as much as possible while holding total resource usage constant.

The tests conducted can be categorized into three groups that investigate (1) the effects of communication speed; (2) the effects of system configurations; and (3) the effects of data sizes. In this section, we first describe the methodology used in the analysis. Then the details of the tests, including the parameter settings and test results are discussed.

6.3.1. Analysis Methodology

For each algorithm, we will compute the following quantities:

$$T \qquad \text{elapsed time}$$
$$T_{cpu} \qquad \text{total CPU time}$$
$$T_{disk} \qquad \text{total disk transfer time}$$
$$T_{net} \qquad \text{total network transfer time}$$

Each of these is the sum over all phases i of the corresponding per-phase quantities T^i, T^i_{cpu}, T^i_{disk}, and T^i_{net}. Resource utilization percentages are easily derived from these basic measures. The analysis uses the following times for basic operations:

$$t_{comp} \qquad \text{CPU time to compare two attributes}$$
$$t_{hash} \qquad \text{CPU time to compute hash function of a key}$$
$$t_{move} \qquad \text{CPU time to move a tuple in memory}$$
$$t_{swapp} \qquad \text{CPU time to swap two pointers in memory}$$
$$t_{build-tuple} \qquad \text{CPU time to build a join result tuple}$$
$$t_{send} \qquad \text{CPU time to send a page over network}$$
$$t_{recv} \qquad \text{CPU time to receive a page over network}$$
$$t_{net} \qquad \text{network hardware page transfer time}$$
$$t_{disk} \qquad \text{disk page transfer time}$$

We first compute the following basic quantities for each phase i:

$$P^i_{disk} \qquad \text{the number of disk transfer pages}$$
$$P^i_{send} \qquad \text{the number of pages sent over the network}$$
$$P^i_{recv} \qquad \text{the number of pages received over the network}$$
$$T^i_{join} \qquad \text{CPU time unrelated to disk or net transfers}$$

P^i_{recv} can be greater than P^i_{send} due to broadcasting. Then,

$$T^i_{cpu} = T^i_{join} + P^i_{send} \cdot t_{send} + P^i_{recv} \cdot t_{recv}$$
$$T^i_{disk} = P^i_{disk} \cdot t_{disk}$$
$$T^i_{net} = P^i_{send} \cdot t_{net}$$

We assume that the CPU time attributable to disk transfers is negligible. For network communication, we consider both the CPU time and the hardware transfer time; either is a potential bottleneck.

The elapsed time T^i for phase i will in general be significantly less than $T^i_{cpu} + T^i_{disk} + T^i_{net}$ due to overlap. It is computed as the minimum of:

- The total disk transfer time of any disk. If I/O is spread evenly over all disks, this quantity is $T^i_{disk}/(NC \cdot ND)$.

- The total network transfer time T^i_{net} for the phase.

- The total CPU time for any single process, including network send and receive processes.

- The total CPU time for any cluster, divided by NP, the number of processors per cluster. If processing is spread evenly over all clusters, this quantity is $T_{cpu}^i/(NC \cdot NP)$. This quantity models processor sharing among the processes at a cluster.

The rationale for this approximation of elapsed time is as follows. In each phase, the processes execute in pipelined fashion. The time for data to flow from the beginning of a pipeline to the end is assumed to be negligible compared to the total elapsed time, so the pipeline is in steady state for most of the phase. Sufficient buffering is provided to permit all processes to execute in parallel with each other and with disk and network transfers:

- One buffer is allocated for each input and output stream for each process.

- One buffer is allocated for each disk to contain the data being read or written.

- One buffer is allocated for the network send process at each cluster to hold the next page to be transmitted, and one buffer for the network receive process to hold the next incoming page.

The detailed formulas derived in the analysis are listed in the next section.

6.3.1.1. Analysis of Join Algorithms in Section 6.2

In this section, we list the detailed formulas used in the analysis of the join algorithms described in 6.3.

6.3.1.1.1. Elapsed Time of Type 1 Sort-Merge Join Algorithm, SMJ1

The parameters that determine the performance of the algorithm include the architecture, timing, and workload parameters listed earlier, and algorithm-specific parameters: $NFRUN_R$, the number of final runs of R, $NFRUN_S$, the number of final runs of S, and NP_{heap}, the number of heap processes per cluster.

There are three major phases in this algorithm: (1) Generate runs of R, (2) Generate runs of S, and (3) Merge R-runs, merge S-runs, and merge-join. The total execution time for the algorithm is therefore $T_{SMJ1} = T_1 + T_2 + T_3$

T_1 and T_2 — Execution Time of Phase 1 and 2. The memory requirements per cluster for this phase are as follows: $ND + NP_{heap}$ input buffers, the same number of output buffers, and NP_{heap} heap areas of the size required to fill the remainder of the M bytes dedicated to the algorithm. Let this size be H bytes; $NHREC_R$ and $NHPAG$ be the number of R records and pages that it can hold. We have

$$H = \frac{M - 2PG \cdot (ND + NP_{heap})}{NP_{heap}},$$

$$NHREC_R = \frac{H}{F \cdot ts(R)},$$

$$NHPAG = \frac{H}{F \cdot PG}$$

There is three processing stage in phase 1: *IRH* (read R from disk until heap filled), *IDRH* (read R from disk and write runs to disk until all of R is read from disk), and *DRH* (empty heap onto disk, completing R-runs). The heap processing time of these stages is calculated as follows:

$$t_{IRH} = TPR \cdot (t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(NHREC_R))$$

$$t_{IDRH} = TPR \cdot (2 \cdot t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(NHREC_R))$$

$$t_{DRH} = TPR \cdot (t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(NHREC_R))$$

The execution time of this phase is:

$$T_1 = \min(NP_{heap} \cdot NHPAG, \frac{|R|}{NC}) \cdot (\max(\frac{t_{disk}}{ND}, \frac{t_{IRH}}{NP_{heap}}) +$$

$$\max(\frac{t_{disk}}{ND}, \frac{t_{DRH}}{NP_{heap}})) +$$

$$\max(\frac{|R|}{NC} - NP_{heap} \cdot NHPAG, 0) \cdot \max(\frac{2 \cdot t_{disk}}{ND}, \frac{t_{IDRH}}{NP_{heap}})$$

The calculation of execution time of phase 2, T_2 is the same as for phase 1, substituting S for R.

T_3 — Execution Time of Phase 3. There are five pipelined processing stages: the first and second merge stages for R and S, and the merge-join, denoted as *M1R*, *M2R*, *M1S*, *M2S*, and *MJ*. Stages *M1R*, *M2R*, and *MJ* are performed at each cluster; stage *M2R* is performed at $NFRUN_R$ clusters, and stage *M2S* is performed at $NFRUN_S$ clusters. Since the runs produced in phase 1 contain an average of $2 \cdot NHPAG$ pages, the merge factor for stage *M1R* is $MF1_R = \dfrac{|R|}{2 \cdot NHPAG \cdot NC}$. The merge factor $MF1_S$ for stage *M1S* is similar. The merge factor for stage *M2R* is $NFRUN_S$. The merge factor for stage *M2S* is $NFRUN_R$.

Each cluster requires two buffers per input stream and two buffers per output stream for stages *M1R*, *M2R*, and *MJ*. Assuming that merging and merge-joining have no other memory requirements, this gives a memory requirement per cluster (in bytes) of $2 \cdot (MF1_R + MF1_S + 4) \cdot PG$. The $NFRUN_R$ clusters that perform *M2R* require an additional $2 \cdot (NFRUN_R + 1) \cdot PG$ bytes, and the $NFRUN_S$ clusters that perform *M2S* require an additional $2 \cdot (NFRUN_S + 1) \cdot PG$ bytes.

The per-page execution times for the stages are:

$$t_{M1R} = TPR \cdot (t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(MF1_R))$$

$$t_{M2R} = TPR \cdot (t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(NFRUN_R))$$

$$t_{M1S} = TPS \cdot (t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(MF1_S))$$

$$t_{M2S} = TPS \cdot (t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(NFRUN_S))$$

$$t_{MJ} = TP_T \cdot \left(\frac{||R||/NFRUN_R + ||S||/NFRUN_S}{||T/NC} \cdot t_{comp} + t_{build-tuple} \right)$$

The network sends and receives performed by the pivot and non-pivot clusters are summarized in the following table:

	# sends	# receives
non-pivot	$\dfrac{|R| + |S|}{NC}$	$\dfrac{|R|}{NFRUN_R} + \dfrac{|S|}{NFRUN_S}$
row pivot	$\dfrac{|S|}{NC} + \dfrac{|R|}{NFRUN_R}$	$(NFRUN_S - 1) \cdot \dfrac{|R|}{NC} + \dfrac{|S|}{NFRUN_S}$
column pivot	$\dfrac{|R|}{NC} + \dfrac{|S|}{NFRUN_S}$	$(NFRUN_R - 1) \cdot \dfrac{|S|}{NC} + \dfrac{|R|}{NFRUN_S}$

The total number of pages transmitted over the network is $(2 - \dfrac{1}{NFRUN_S}) \cdot |R| + (2 - \dfrac{1}{NFRUN_R}) \cdot |S|$ This is less than the total number of pages received because the pivot clusters use a one-to-many broadcast to send their results to the merge-join processes.

The total execution time for phase 3 is therefore

$$T_3 = \max\left(\frac{|R| + |S|}{NC \cdot ND} \cdot t_{disk}, \ \left((2 - \frac{1}{NFRUN_S}) \cdot |R| + (2 - \frac{1}{NFRUN_R}) \cdot |S| \right) \cdot t_{net}, \right.$$

$$\max\left(\frac{|S|}{NC} + \frac{|R|}{NFRUN_R}, \ \frac{|R|}{NC} + \frac{|S|}{NFRUN_S} \right) \cdot t_{send},$$

$$\left(\frac{|R|}{NFRUN_R} + \frac{|S|}{NFRUN_S} \right) \cdot t_{recv},$$

$$\left. \frac{|R|}{NC} \cdot t_{M1R}, \ \frac{|S|}{NC} \cdot t_{M1S}, \ \frac{|R|}{NFRUN_R} \cdot t_{M2R}, \ \frac{|S|}{NFRUN_S} \cdot t_{M2S}, \ \frac{|T|}{NC} \cdot t_{MJ} \right)$$

6.3.1.1.2. Elapsed Time of Type 2 Sort-Merge Join Algorithm, SMJ2

There are also three major phases in algorithm SMJ2: (1) Generate initial runs of R and store on disk, (2) Merge R-runs into complete sorted relations, broadcast to all clusters, and (3) Generate S-runs and merge-join with R.

Two algorithm-specific parameters are: NP_{heap}, the number of heap processes used in phase 1 to generate the initial runs of R, and NP_{mj}, the number of heap and merge-join process pairs in phase 3.

T_1 — the execution time of phase 1, is identical to that of phase 1 for the type 1 sort merge join algorithm.

T_2 — Execution Time of Phase 2. There are two processing stages: the first and second merge stages, denoted $M1R$ and $M2R$. $M1R$ is performed at all clusters, and $M2R$ is performed at C_1. Clusters other than C_1 require a process to receive R from the network and write it to disk. The merge factor for $M1R$ is (as before) $MF1_R = \dfrac{|R|}{2 \cdot NHPAG \cdot NC}$, and the merge factor for $M2R$ is NC.

Inputs and outputs are double buffered. Clusters other than C_1 execute $M1R$ and the net-to-disk copy, so they require $2 \cdot (MF1_R + 3) \cdot PG$ bytes. Cluster C_1 requires an additional $2 \cdot (NC + 1) \cdot PG$ bytes for stage $M2R$. The per-page execution times are:

$$t_{M1R} = TPR \cdot (t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(MF1_R))$$

$$t_{M2R} = TPR \cdot (t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(NC))$$

At each cluster, $M1R$ produces $|R|/NC$ pages. $M2R$ produces $|R|$ pages at C_1. Each cluster reads $|R|/NC$ pages and writes $|R|$ pages. A total of $(2-1/NC) \cdot |R|$ pages are sent over the network. The total execution time for this phase is therefore

$$T_2 = \max((1+\frac{1}{NC}) \cdot \frac{|R|}{ND} \cdot t_{disk}, \; (2-\frac{1}{NC}) \cdot |R| \cdot t_{net}, \; |R| \cdot t_{send}, \; |R| \cdot t_{recv},$$
$$\frac{|R|}{NC} \cdot t_{M1R}, \; |R| \cdot t_{M2R})$$

T_3 — Execution Time of Phase 3. The heap and merge-join stages of this phase are denoted $IDSH$ and MJ respectively. Assuming NP_{mj} heap processes and merge-join processes at each cluster, the per-cluster memory requirements are as follows: $(ND + NP_{mj}) \cdot PG$ bytes for input buffers to $IDSH$, $2 \cdot NP_{mj} \cdot PG$ bytes for buffers between $IDSH$ and MJ processes, a like amount for input buffers to MJ for relation R, and $NP_{mj} \cdot H_S$ bytes for the heaps, where H_S is defined so that available memory is fully utilized: H_s and the number of record and pages of S that can fit in each heap are, respectively:

$$H_S = \frac{M - ND \cdot PG}{NP_{mj}} - 5 \cdot PG, \qquad NHREC_S = \frac{H_S}{F \cdot ts(S)}, \qquad NHPAG_S = \frac{H_S}{F \cdot PG}$$

The following analysis is simplified by assuming that the I/O rate is constant over the phase. In fact, R is read only after the heaps are filled.

The number of disk reads per cluster is

$$\frac{|S|}{NC} + \frac{|S|}{2 \cdot NP_{mj} \cdot NHPAG_S \cdot NC} \cdot |R|$$

There is no network I/O. The per-page processing times for the two stages are:

$$t_{IDSH} = TPS \cdot (2 \cdot t_{move} + (2 \cdot t_{comp} + t_{swapp}) \cdot \lg(NHREC_S))$$

$$t_{MJ} = TP_T \cdot \left(\frac{||R|| \cdot \dfrac{|S|}{2 \cdot NHPAG_S} + ||S||}{||T||} \cdot t_{comp} + t_{build-tuple} \right)$$

The execution time for this phase is

$$T_3 = \max\left(\left(\frac{|S|}{NC} + \frac{|S|}{2 \cdot NP_{mj} \cdot NHPAG_S \cdot NC} \cdot |R|\right) \cdot t_{disk}, \quad \frac{|S|}{NP_{mj} \cdot NC} \cdot t_{IDSH},\right.$$

$$\left. \frac{|T|}{NP_{mj} \cdot NC} \cdot t_{MJ}\right)$$

6.3.1.1.3. Analysis of Algorithms

In the following sections we list the formulas used in evaluting hash-based join algorithms, which compute the number of disk I/O and network transfer and the CPU processing time for each phase. The phases 3 and 4 for hybrid hash algorithms and the phases of partitioning and joining batches 1 to $NBATCH$ are the same, the quantities listed in the formulas are the totals of all these batches. The elapsed time is computed from the number of disk I/O's, the number of pages transferred, and the CPU time obtained using the method described in Section 6.3.1.

The Number of Batches and Batch Sizes. Before giving the formulas, we first briefly describe the computation of the number of batches and the data size of the batches, $NBATCH$, $|RB_0|$, $||RB_0||$, $|SB_0|$, $||SB_0||$, etc. These quantities are determined by the amount of memory left in each cluster after allocating buffers for disk and network I/O and interprocess communication according to the principles described in the above section. They are computed as follows:

Let NBF_0 be the number of buffer pages required during partitioning and NBF_1 be the number of buffer pages required at other times. Let us first compute NBF_1, which is algorithm-specific. For the first type of hash-based algorithms HSM1 and HH1, where both relations are transferred among clusters, each join processor has three output streams after reading in and hashing tuples: tuples to be used to probe the hash table for itself, tuples to be used for other processors at the same cluster to probe the corresponding hash tables, and tuples to send to other clusters. That is, three kinds of buffer pages have to be allocated for each processor. The total number of buffer pages is therefore

$$NBF_1 = NC \cdot (ND + 2 + NP_{join} \cdot (NC + NP_{join} - 1))$$

For another two algorithms, HSM2 and HH2, where the small relation is replicated at every cluster, all tuples arriving at a join processor will either be processed by the processor itself, or processors at the same cluster, less buffer pages are needed.

For HSM2 and HH2,

$$NBF_1 = NC \cdot (ND + 2 + NP_{join} \cdot (NP_{join} + 1))$$

In the partition phase, each processor also generates a stream of tuples which are written back to disk for later processing, NBF_0 is therefore related to NBF_1 as follows:

$$NBF_0 = NBF_1 + NBATCH \cdot NC \cdot NP_{join}$$

The size of the batches can be computed as follows. Let NDP be the total number of data pages staged in main memory during join processing for sorting (HSM1 and HSM2) of building hash tables (HH1 and HH2), excluding buffers. For algorithms HSM1 and HSM2, $NDP = |R| + |S|$, since these algorithms stage batches of both relations prior to joining them. For algorithms HH1 and HH2, $NDP = |R|$. Let NDP_0 be the number of data pages staged to join batch 0 and NDP_1 be the number of data pages staged for each subsequent batch. Clearly,

$$NDP = NDP_0 + NBATCH \cdot NDP_1$$

We want to maximize NDP_0 to minimize the amount of intermediate data written to disk. We therefore assign all available memory to stage batch 0:

$$NBF_0 + F \cdot NDP_0 \leq NC \cdot M$$

with the equality holding unless $NBATCH = 0$. Here, F represents the "universal fudge factor", a number slightly greater than one that accounts for the memory overhead required for a hash table or binary search tree of tuples, beyond the memory required for the tuples themselves. To maximize NDP_0, we must minimize NBF_0, and therefore $NBATCH$, subject to the constraint

$$NBF_1 + F \cdot NDP_1 \leq NC \cdot M$$

It is relatively easy to derive the following:

$$NBATCH = \max(0, \left\lceil \frac{F \cdot NDP + NBF_1 - NC \cdot M}{NC \cdot M - NBF_1 - NC \cdot NP_{join}} \right\rceil)$$

With $NBATCH$ and the above equations, we can then calculate the data size $|RB_0|$, $|SB_0|$, etc.

6.3.1.1.4. Analysis of Algorithm HSM1

Partition Batch 0:

$P_{disk}:$ $2(|R| + |S|) - |RB_0| - |SB_0|$ read R and S, write batches

$P_{send}:$ $\dfrac{NC-1}{NC}(|RB_0| + |SB_0|)$

P_{recv}: $\frac{NC-1}{NC}\cdot(|RB_0|+|SB_0|)$

T_{join}: $(||R||-||RB_0||+||S||-||SB_0||)\cdot(t_{hash}+t_{move})$ tuples to disk buffer

$\quad+\frac{NC-1}{NC}\cdot(||RB_0||+||SB_0||)\cdot(t_{hash}+t_{move})$ tuples to network buffer

$\quad+\frac{NP-1}{NP}\cdot(||RB_0||+||SB_0||)\cdot(t_{hash}+t_{move})$ tuples to interprocess buffer

$\quad+||RB_0||\cdot(\lg\frac{||RB_0||}{NP_{join}\cdot NC}\cdot t_{comp}+t_{move})$ insert R tuples

$\quad+||SB_0||\cdot(\lg\frac{||SB_0||}{NP_{join}\cdot NC}\cdot t_{comp}+t_{move})$ insert S tuples

Join Batch 0:

T_{join}: $2\cdot(||RB_0||+||SB_0||)\cdot t_{comp}$ find matching tuples

$\quad+JS\cdot||R||\cdot||SB_0||\cdot t_{build-tuple}$ generate result tuples

Partition Batches $1-NBATCH$:

P_{disk}: $|R|-|RB_0|+|S|-|SB_0|$ read batches

P_{send}: $\frac{NC-1}{NC}\cdot(|R|-|RB_0|+|S|-|SB_0|)$

P_{recv}: $\frac{NC-1}{NC}\cdot(|R|-|RB_0|+|S|-|SB_0|)$

T_{join}: $\frac{NC-1}{NC}\cdot(||R||-||RB_0||+||S||-||SB_0||)\cdot(t_{hash}+t_{move})$ tuples to network buffer

$\quad+\frac{NP-1}{NP}\cdot(||R||-||RB_0||+||S||-||SB_0||)\cdot(t_{hash}+t_{move})$ tuples to interprocess buffer

$\quad+(||R||-||RB_0||)\cdot(\lg\frac{||R||-||RB_0||}{NP_{join}\cdot NC\cdot NBATCH}\cdot t_{comp}+t_{move})$ insert R tuples

$\quad+(||S||-||SB_0||)\cdot(\lg\frac{||S||-||SB_0||}{NP_{join}\cdot NC\cdot NBATCH}\cdot t_{comp}+t_{move})$ insert S tuples

Join Batches $1 - NBATCH$:

T_{join}: $\quad 2\cdot(\|R\| - \|RB_0\| + \|S\| - \|SB_0\|)\cdot t_{comp}$ find matching tuples

$\quad + JS\cdot\|R\|\cdot(\|S\| - \|SB_0\|)\cdot t_{build-tuple}$ generate result tuples

6.3.1.1.5. Analysis of Algorithm HSM2

Partition Batch 0:

P_{disk}: $\quad 2\cdot(|R| + |S|) - |RB_0| - |SB_0|$ read R and S, write batches

P_{send}: $\quad |RB_0|$

P_{recv}: $\quad (NC-1)\cdot|RB_0|$

T_{join}: $\quad \|R\|\cdot(t_{hash} + t_{move})$ tuples to disk or broadcast buffer

$$+ NC\cdot\|RB_0\|\cdot(\lg\frac{\|RB_0\|}{NP_{join}\cdot NC}\cdot t_{comp} + t_{move})$$ insert R tuples

$$+ \|SB_0\|\cdot(\lg\frac{\|SB_0\|}{NP_{join}\cdot NC}\cdot t_{comp} + t_{move})$$ insert S tuples

Join Batch 0:

T_{join}: $\quad 2\cdot(NC\cdot\|RB_0\| + \|SB_0\|)\cdot t_{comp}$ find matching tuples

$\quad + JS\cdot\|R\|\cdot\|SB_0\|\cdot t_{build-tuple}$ generate result tuples

Partition Batches $1 - NBATCH$

P_{disk}: $\quad |R| - |RB_0| + |S| - |SB_0|$ read batches

P_{send}: $\quad |R| - |RB_0|$

P_{recv}: $\quad (NC-1)\cdot(|R| - |RB_0|)$

T_{join}: $\quad (\|R\| - \|RB_0\|)\cdot(t_{hash} + t_{move})$ tuples to broadcast buffer

$$+ NC\cdot(\|R\| - \|RB_0\|)\cdot(\lg\frac{\|R\| - \|RB_0\|}{NP_{join}\cdot NC\cdot NBATCH}\cdot t_{comp} + t_{move})$$ insert R tuples

$$+(||S||-||SB_0||)\cdot(\lg\frac{||S||-||SB_0||}{NP_{join}\cdot NC\cdot NBATCH}\cdot t_{comp}+t_{move})$$

insert S tuples

Join Batches $1-NBATCH$:

T_{join}: $\qquad 2\cdot(NC\cdot(||R||-||RB_0||)+||S||-||SB_0||)\cdot t_{comp}$

find matching tuples

$$+JS\cdot||R||\cdot(||S||-||SB_0||)\cdot t_{build-tuple}$$

generate result tuples

6.3.1.1.6. Analysis of Algorithm HH1

Phase 1:

P_{disk}: $\qquad 2\cdot|R|-|RB_0|$

read R, write batches

P_{send}: $\qquad \dfrac{NC-1}{NC}\cdot|RB_0|$

P_{recv}: $\qquad \dfrac{NC-1}{NC}\cdot|RB_0|$

T_{join}: $\qquad (||R||-|RB_0|)\cdot(t_{hash}+t_{move})$

tuples to disk buffer

$$+\frac{NC-1}{NC}\cdot||RB_0||\cdot(t_{hash}+t_{move})$$

tuples to network buffer

$$+\frac{NP-1}{NP}\cdot||RB_0||\cdot(t_{hash}+t_{move})$$

tuples to interprocess buffer

$$+||RB_0||\cdot(t_{hash}+t_{move})$$

tuples to hash table

Phase 2:

P_{disk}: $\qquad 2\cdot|S|-|SB_0|$

read S, write batches

P_{send}: $\qquad \dfrac{NC-1}{NC}\cdot|SB_0|$

P_{recv}: $\qquad \dfrac{NC-1}{NC}\cdot|SB_0|$

T_{join}: $\qquad (||S||-||SB_0||)\cdot(t_{hash}+t_{move})$

tuples to disk buffer

$$+ \frac{NC-1}{NC} \cdot ||SB_0|| \cdot (t_{hash} + t_{move})$$ tuples to network buffer

$$+ ||SB_0|| \cdot (t_{hash} + F \cdot t_{comp})$$ probe hash table

$$+ JS \cdot ||R|| \cdot ||SB_0|| \cdot t_{build-tuple}$$ generate result tuples

Phase 3:

P_{disk}: $|R| - |RB_0|$

P_{send}: $\dfrac{NC-1}{NC} \cdot (|R| - |RB_0|)$

P_{recv}: $\dfrac{NC-1}{NC} \cdot (|R| - |RB_0|)$

T_{join}: $(\dfrac{NC-1}{NC} + \dfrac{NP-1}{NP} + 1) \cdot (||R|| - ||RB_0||) \cdot (t_{hash} + t_{move})$

Phase 4:

P_{disk}: $|S| - |SB_0|$

P_{send}: $\dfrac{NC-1}{NC} \cdot (|S| - |SB_0|)$

P_{recv}: $\dfrac{NC-1}{NC} \cdot (|S| - |SB_0|)$

T_{join}: $\dfrac{NC-1}{NC} \cdot (||S|| - ||SB_0||) \cdot (t_{hash} + t_{move})$
$$+ (||S|| - ||SB_0||) \cdot (t_{hash} + F \cdot t_{comp})$$
$$+ JS \cdot ||R|| \cdot (||S|| - ||SB_0||) \cdot t_{build-tuple}$$

6.3.1.1.7. Analysis of Algorithm HH2

Phase 1:

P_{disk}: $2 \cdot |R| - |RB_0|$ read R, write batches

P_{send}: $|RB_0|$

P_{recv}: $(NC-1) \cdot |RB_0|$

T_{join}: $||R|| \cdot (t_{hash} + t_{move})$ tuples to disk or broadcast buffer

$+ NC \cdot ||RB_0|| \cdot (t_{hash} + t_{move})$ tuples to hash table

Phase 2:

P_{disk}: $2 \cdot |S| - |SB_0|$ read S, write batches

T_{join}: $(||S|| - ||SB_0||) \cdot (t_{hash} + t_{move})$ tuples to disk buffer

$+ ||SB_0|| \cdot (t_{hash} + F \cdot t_{comp})$ probe hash table

$+ JS \cdot ||R|| \cdot ||SB_0|| \cdot t_{build-tuple}$ generate result tuples

Phase 3:

P_{disk}: $|R| - |RB_0|$

P_{send}: $|R| - |RB_0|$

P_{recv}: $(NC - 1) \cdot (|R| - |RB_0|)$

T_{join}: $(||R|| - ||RB_0||) \cdot (t_{hash} + t_{move})$ tuples to broadcast buffer

$+ NC \cdot (||R|| - ||RB_0||) \cdot (t_{hash} + t_{move})$ tuples to hash table

Phase 4:

P_{disk}: $|S| - |SB_0|$

T_{join}: $(||S|| - ||SB_0||) \cdot (t_{hash} + F \cdot t_{comp})$

$+ JS \cdot ||R|| \cdot (||S|| - ||SB_0||) \cdot t_{build-tuple}$

6.3.2. Parameter Settings

Three types of parameters are used in the comparisons: architectural parameters, timing parameters, and workload parameters. The parameter values used are listed below.

	Parameter	Range	Typical Value
Architectural Parameters	NC	2 - 64	16
	ND	2 - 20	8
	NP	2 - 64	8
	M	16 - 512	128
	PG	32K bytes	
Timing Parameters	t_{comp}	5 μs	
	t_{hash}	3 μs	
	t_{move}	10 μs	
	t_{swapp}	1μs	
	$t_{build-tuple}$	20 μs	
	t_{send}	0.1 - 3 ms	1 ms
	t_{recv}	0.1 - 3 ms	1 ms
	t_{net}	100 - 600 Mbps	100 Mbps
	t_{disk}	10 - 30 ms	15 ms
Workload Parameters	$\mid R \mid$	50 - 400K pages	100K pages
	$\mid S \mid$	$\mid R \mid$ - $10 \cdot \mid R \mid$	$\mid R \mid$
	JS	10^{-8}	

6.3.3. Tests and Results

Now we describe the tests conducted in the performance comparisons. Initial analysis showed that the sort merge algorithms were generally much slower that the hash-based algorithms. Therefore we show the results for the hash-based algorithms only. Figures 6.2-6.10 show the results.

6.3.3.1. Communication versus Performance

In this test, the bandwidth of the communications line was varied from 100 Mbps to 600 Mbps to study the effects of the data transfer rate on the performance of the algorithms. The results are shown in figure 6.2. The elapsed times of type 1 algorithms drop dramatically when the bandwidth increases from 100 Mbps to 300 Mbps. This is because the system is network bound with our typical parameter settings for these algorithms. In other words, the data transfer was the bottleneck and the bandwidth of the communications line determined their elapsed times. In contrast, the elapsed times of type 2 algorithms did not change at all when the bandwidth was varied in the range. In these two algorithms, the amount of data transferred equals to the size of the small relation R. It was not the bottleneck when the bandwidth is greater than 200 Mbps.

6.3.3.2. System Configuration versus Performance

The first group of tests that investigated the effects of system configurations on the performance consists of the following five tests.

- The total hardware cost, that is, the total number of disks ($NC \cdot ND$), and processors ($NC \cdot NP$), and memory size ($NC \cdot M$) is kept as constants. The number of clusters in the system (NC) is varied.

- The configuration of each cluster is kept the same, (e. g. fix NP and ND), the number of the clusters in the system varies.

- The number of disks at each cluster, ND, is varied and other parameters are kept as constants.

- The number of processors at each cluster, NP, is varied and other parameters are kept as constants.

- Memory size M at each cluster is varied while other parameters were kept as constants.

We will describe these tests in more detail.

(1) *The elapsed time versus different configurations under the same total hardware cost.* That is, the total number of processors and disks, and the total size of memory banks in the system were kept as a constant. The number of clusters were varied and the number of disks and processors and the size of memory bank at each cluster were varied accordingly to keep the total resources constant. One extreme of the spectrum is that all resources form a single large cluster. Another extreme is a system such as Gamma [DeWi86] where each cluster has only one processor, one disk and one piece of memory. The results are shown in figure 6.3. The second extreme case is not shown in the figure since the trend is already shown when the system consists of 64 clusters. That is the case each cluster has two disks and two processors. When the system consists of 128 clusters with one processor and one disk, the elapsed time for HSM-2 is almost doubled compared to 64 cluster case. Other 3 curves kept flat (not shown in the figure).

From figure 6.3, it can be seen that a huge single cluster provides the best performance since the communications cost is eliminated. The curves of HH-2 and HSM-2 are flat since the system is network-bound. This will be seen more clearly later. The total amount of data transferred in the type 2 algorithms equals the size of relation R, which is a constant when the system configuration is changed. With the large number of clusters (> 32), HSM-2 performs poorly since replicating relation R increase the total processing cost. The increasing CPU cost makes the system CPU bound. When the number of clusters is doubled, the elapsed time is also doubled. For the type 1 algorithms, increasing the number of clusters increases the quantity $\dfrac{NC-1}{NC} \cdot (||R|| + ||S||)$, which is the quantity of data transmitted over the network. When the number of clusters in the system increases from 2 to 4, this amount increases one third. This is reflected in the increase of the elapsed time. We ignore the memory contention and the cost of synchronizing the concurrent access of disks in our analysis. The processing power with regard to disk I/O and CPU processing do not change since the number of disks and processors are kept constants no matter how they are organized into clusters during all the

tests. The huge single cluster case is just an indication of the lower bound of the elapsed time. It is impractical to put a large number of disks and processors with shared memory in one cluster.

Figure 6.3 was obtained with 100 Mbps network. With 600 Mbps. the performance is a little different. In this case. the type 1 algorithms performed better than their counterparts. Figure 6.4 shows the results. As in Figure 6.3. the type 2 algorithms performed very poorly when the number of clusters exceeded 64.

(2) *The elapsed time versus the number of clusters.* In this test. the configuration of each cluster was kept the same and the number of clusters in the system was varied. The result of this variation is to increase the parallel processing power of the system and also introduce more data transfer for some algorithms since we assume that the original data is scattered around the system. The result of this test is shown in figure 6.5.

(3) *The elapsed time versus the number of disks at each cluster.* In these tests. the number of disks is varied and other parameters are kept as constant. Figure 6.6 shows the result. It can be seen from the figure that it is unnecessary to attach more disks to a cluster when the bottleneck is not disk I/O. When the number of disks was more than 8 in the tested case. increase of the number of disks did not bring real performance benefit.

(4) *The elapsed time versus the number of processors at each cluster.* In these tests. the number of processors at each cluster was varied and the results are shown in figure 6.7. It can be seen from the figure that CPU processing was not the bottleneck even with 2 processors at each cluster. Only exception was the HSM-2 algorithm which needed the most extensive CPU computation among these algorithms. However. with more than 8 processors per cluster. the elapsed time did not decrease further when more processors were added to the clusters. Another observation is that. in our buffer allocation scheme. the number of buffers needed increases proportionally to the square of number of processors (not linear). The large number of processors may cause insufficient memory for executing the algorithms.

(5) *The elapsed time versus the size of memory bank at each cluster.* In this group of tests. the size of the memory bank at each cluster was varied. From the results. shown in figure 6.8. it can be seen that the type 2 algorithms required more memory space for buffers. That is. the minimum memory requirement is more strict for them. However, as long as the memory was big enough to start the algorithm. there was not a big difference in the elapsed time with different memory sizes. This can be explained as follows. The only benefit a large size memory provide is to save the disk I/O and related rehashing of the first batch of buckets. The processing of the remaining buckets will not be affected by the bucket sizes which are determined by memory size. If the processing cost of the first batch is not the dominant factor of the total processing. or the disk I/O cost is not the bottleneck in the first phase. the memory size will not affect the elapsed time a lot as seen from the figure.

6.3.3.3. Data Sizes versus Performance

The third group of tests studied the effects of the data size on performance of the algorithms. The size of relation R ranged from $1.75 \cdot 10^9$ bytes to $25 \cdot 10^9$. $||S||$ ranged from $||R||$ to $10 \cdot ||R||$. Figure 6.9 depicts the relationship between the elapsed time and the relation size. Along with the increase of the size of two relations, the elapsed time of all algorithms increased. However, the type 1 algorithms were more sensitive to this increase. The elapsed time increased linearly when the size of relations increased. The reason for this is that the bottleneck in these tests are network. The amount of data transferred increases when the relation sizes increase. Figure 6.10 shows the same system and relation sizes with high bandwidth network (600 Mbps). The first observation is that the type 1 algorithms outperform the type 2 algorithms when the relations were small. Second, the elapsed time of all algorithms increases to some extent when the relations become larger. The type 1 algorithms were still more sensitive to the relation sizes. When the relation sizes become larger, their performance become worse than the type 2 algorithms.

Elapsed Time Versus Network Bandwidth
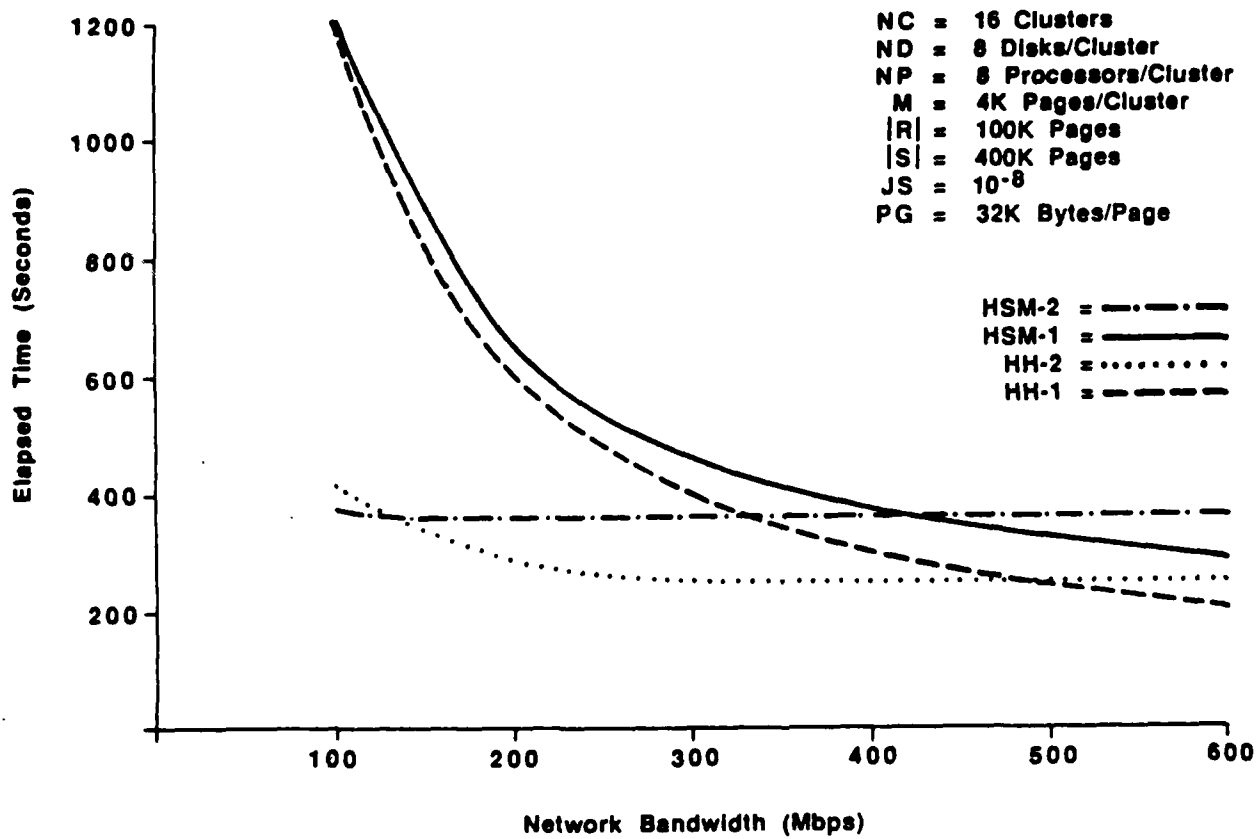
Figure 6.2. Elapsed Time versus Network Bandwidth

Elapsed Time Versus Number Clusters
(Fixed Hardware Cost, Slow Network)
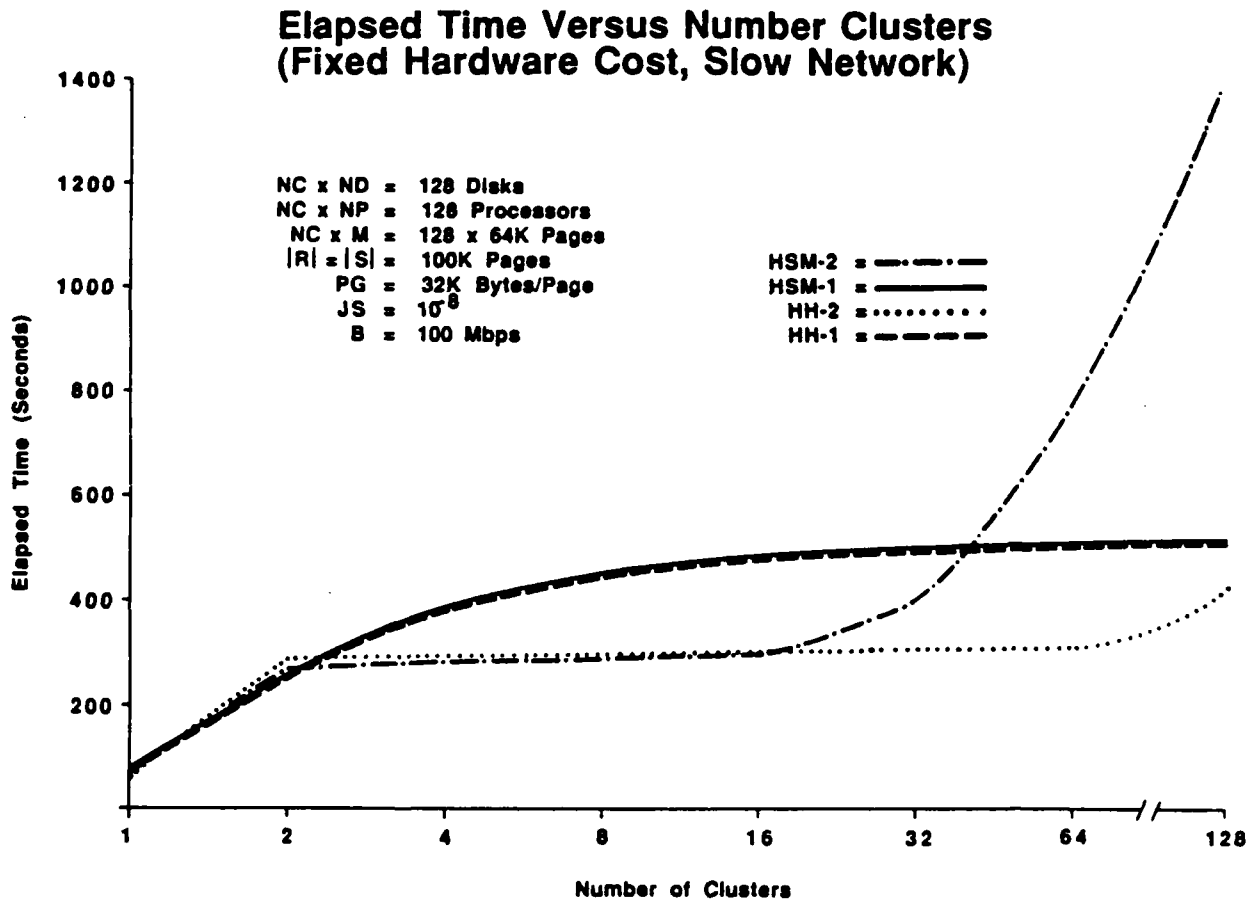
Figure 6.3. Elapsed Time versus Number of Clusters (Fixed Hardware Costs)
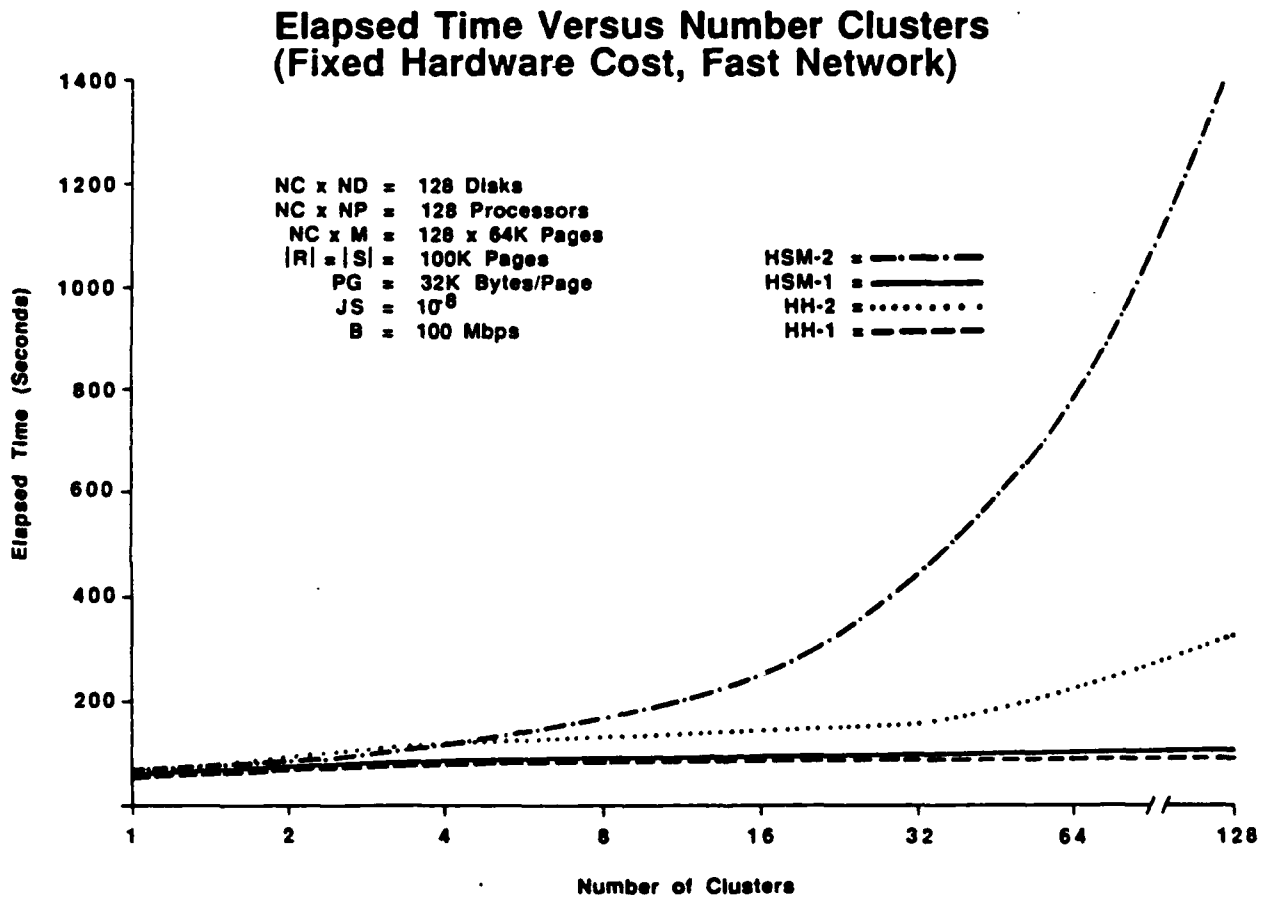
Figure 6.4. Elapsed Time versus Number of Clusters (Fixed Hardware Costs)

Elapsed Time Versus Number Clusters
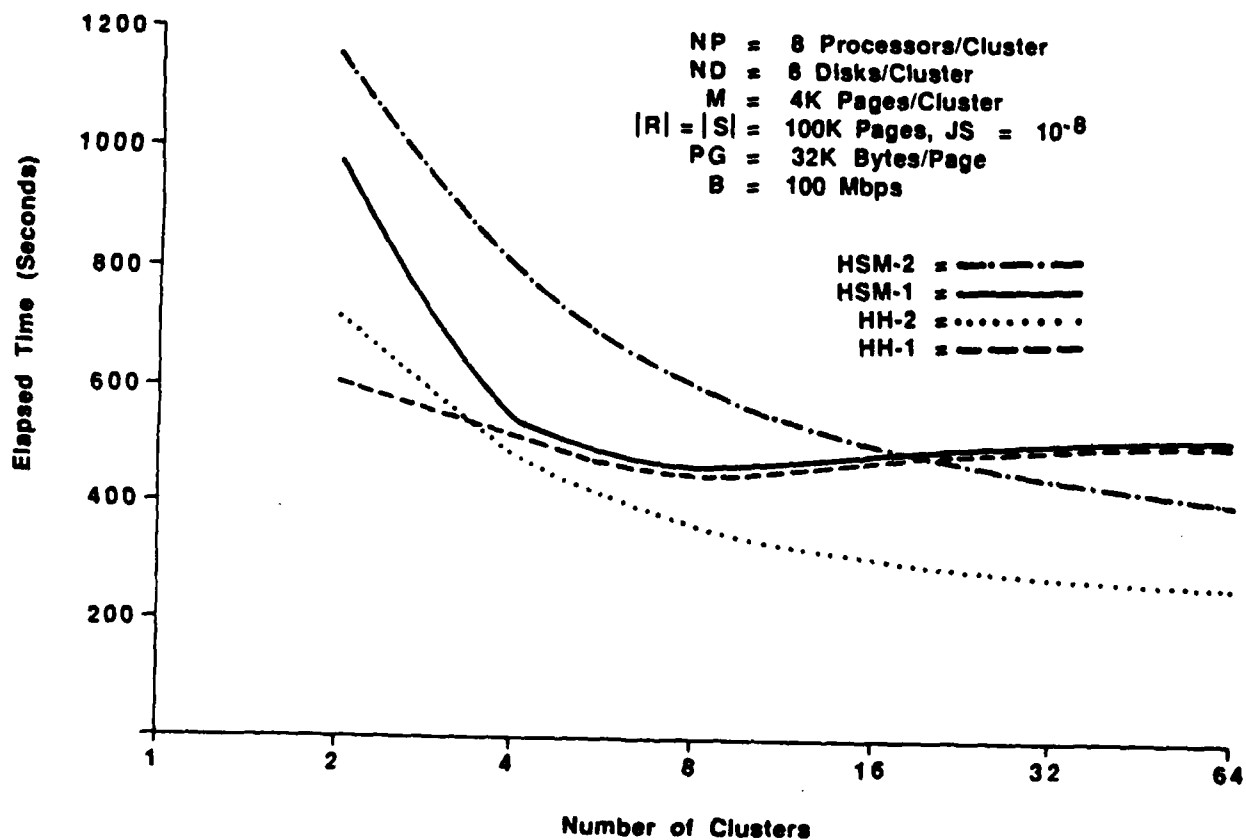
Figure 6.5. Elapsed Time versus Number of Clusters (Fixed NP, ND, and M)

Elapsed Time Versus Number of Disks

NC = 16 Clusters
NP = 8 Processor/Cluster
M = 4K Pages/Cluster
$|R| = |S| =$ 100K Pages
JS = 10^{-8}
PG = 32K Bytes/Page
B = 100 Mbps

HSM-2 = — — — —
HSM-1 = ———
HH-2 = ••••••••
HH-1 = — — — —

Number of Disks/Cluster
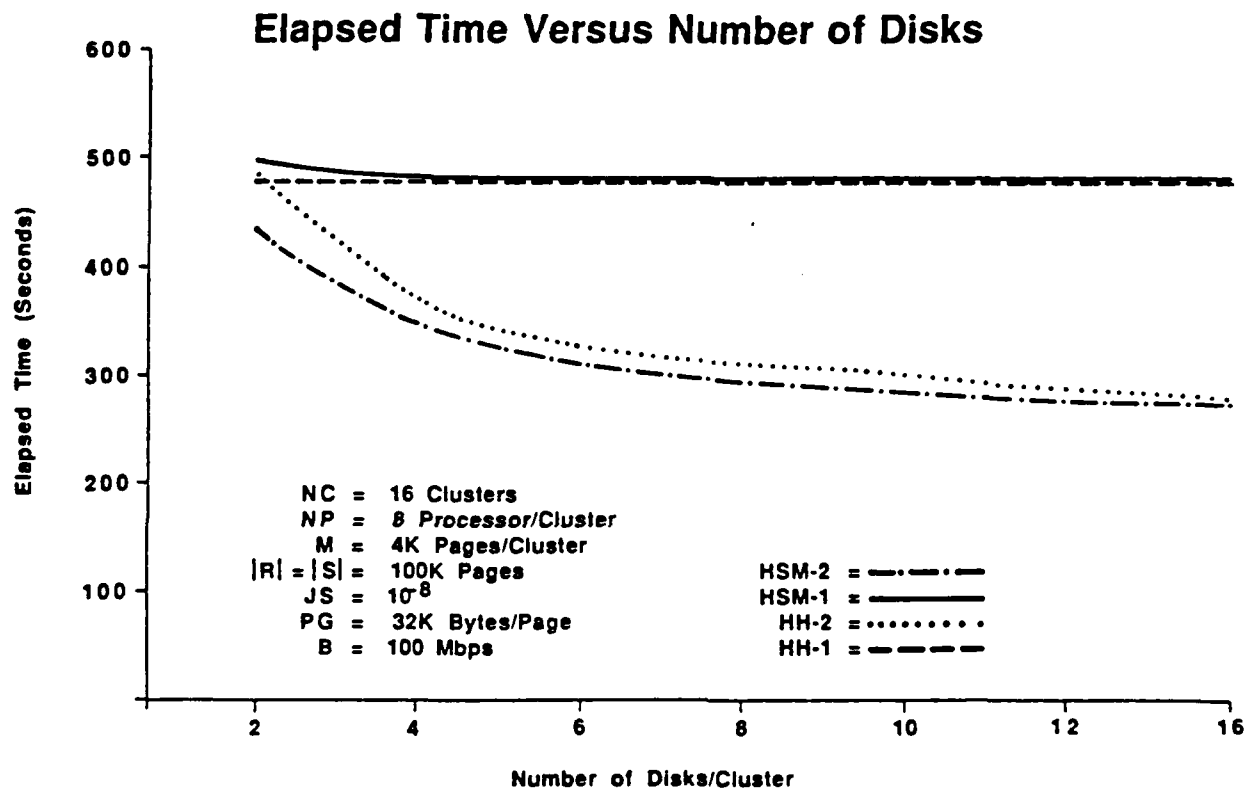
Figure 6.6. Elapsed Time versus Number of Disks

Elapsed Time Versus Number of Processors

NC $=$ 16 Clusters
NP $=$ 8 Processor/Cluster
M $=$ 4K Pages/Cluster
$|R| = |S| =$ 100K Pages
JS $=$ 10^{-8}
PG $=$ 32K Bytes/Page
B $=$ 100 Mbps

HSM-2 $=$
HSM-1 $=$
HH-2 $=$
HH-1 $=$

Elapsed Time (Seconds)

Number of Processors/Cluster

Figure 6.7. Elapsed Time versus Number of Processors

Elapsed Time Versus Memory Size
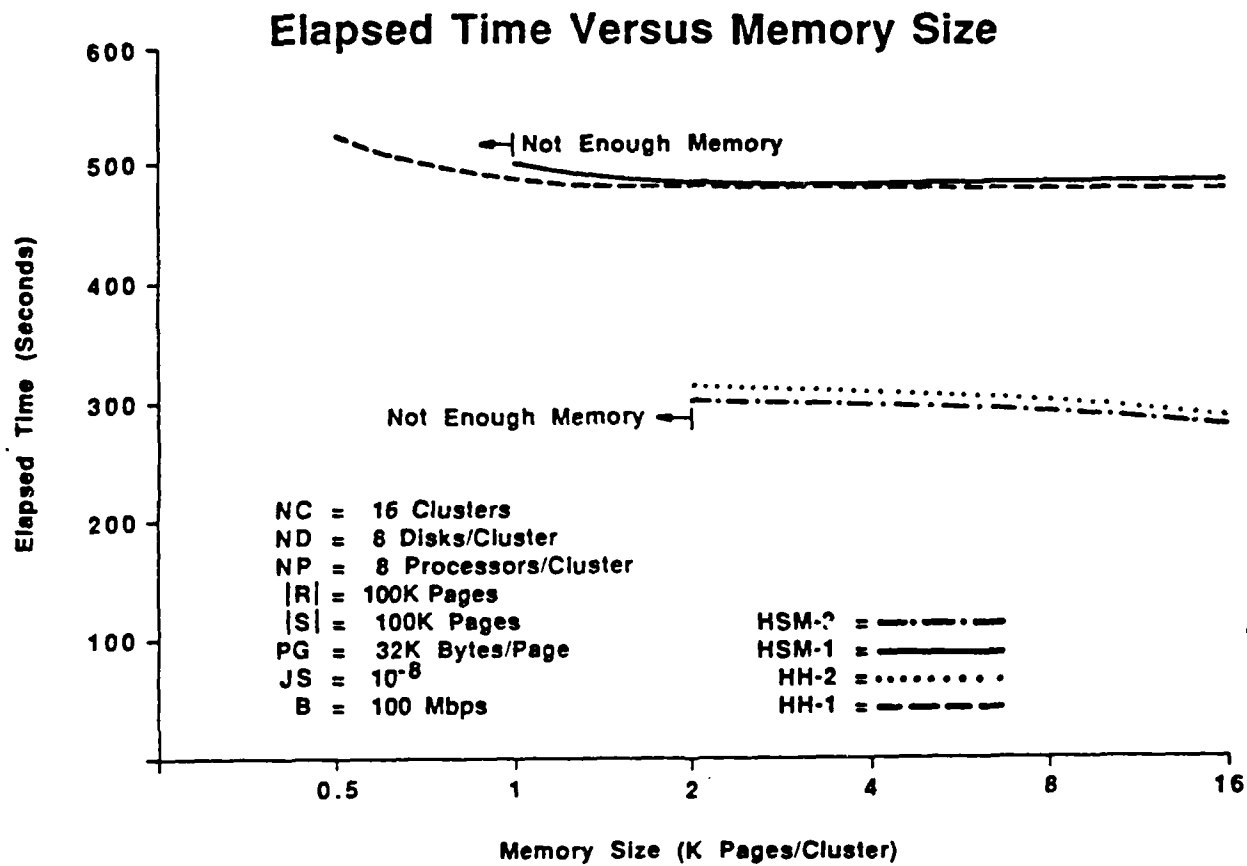
Figure 6.8. Elapsed Time versus Memory Size

Elapsed Time Versus Data Size (B=100 Mbps)
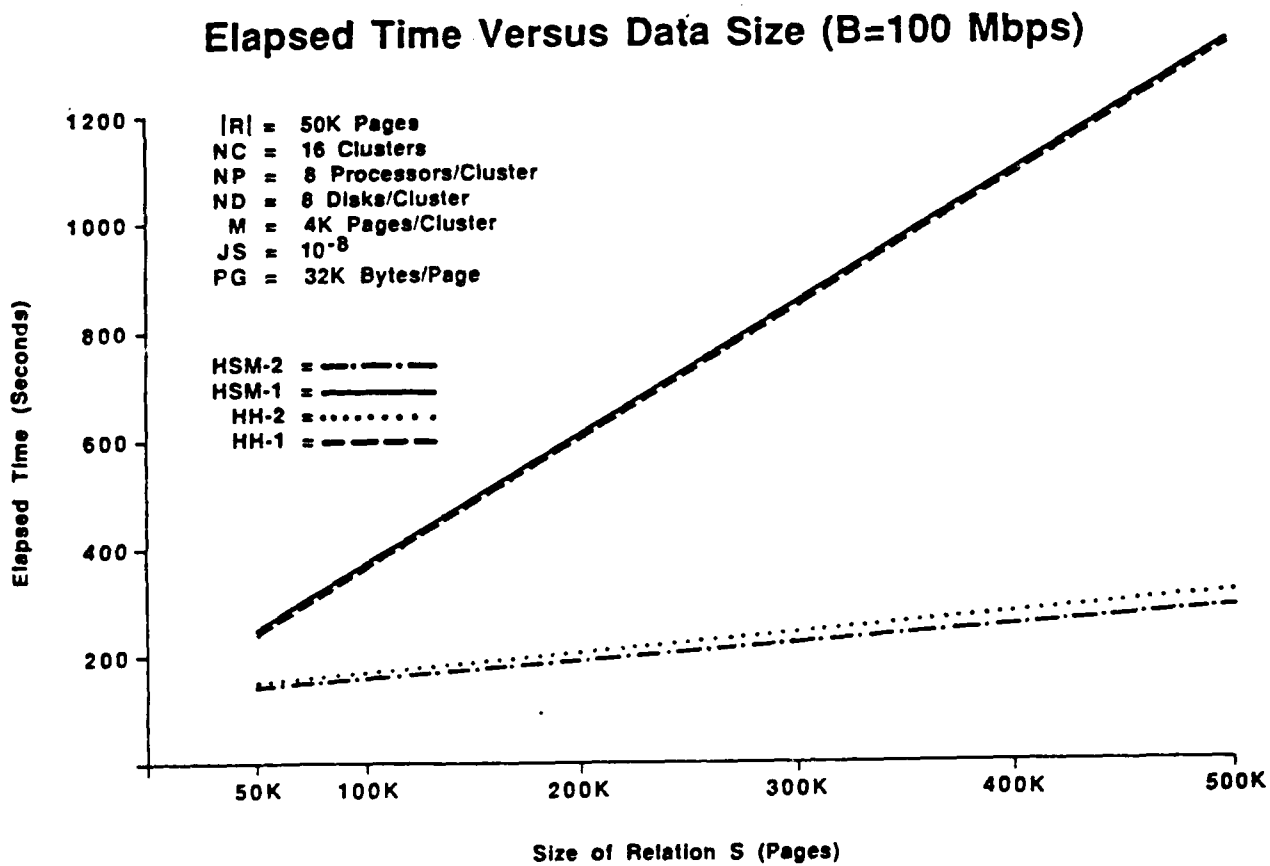
Figure 6.9. Elapsed Time versus Relation Sizes

Elapsed Time Versus Data Size (B=600 Mbps)
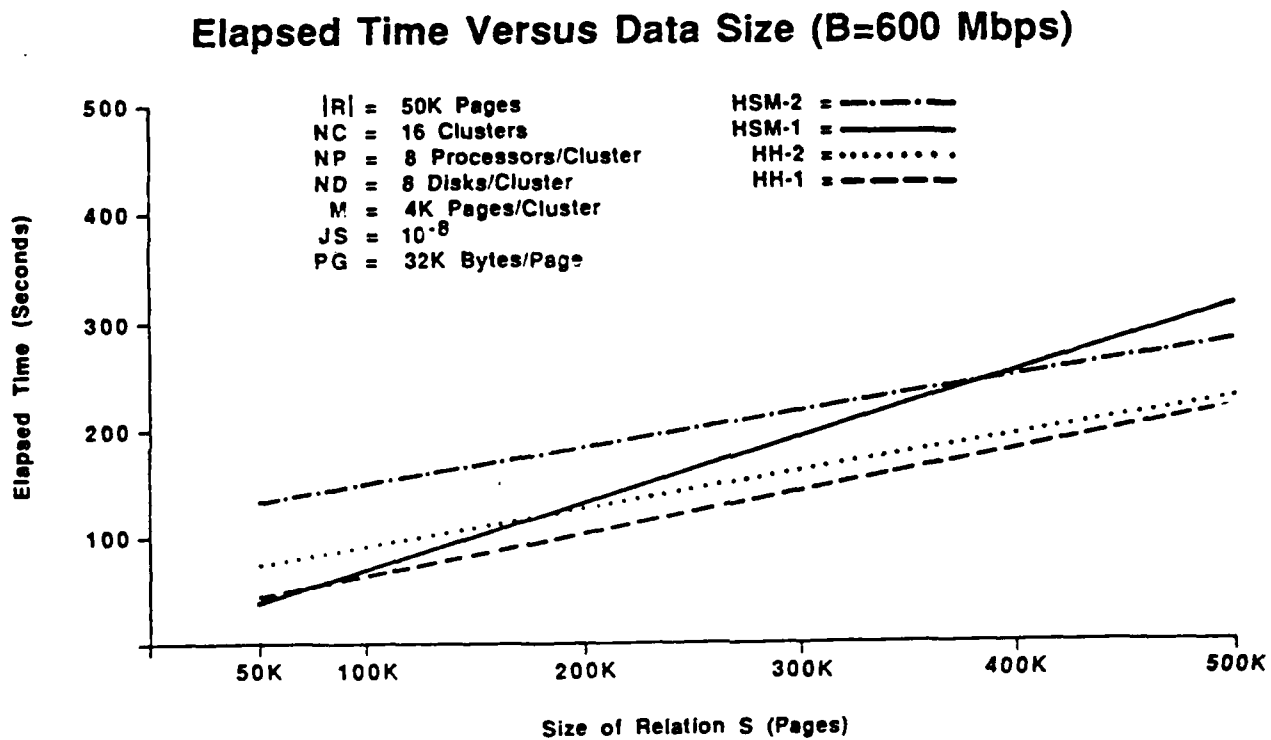
Figure 6.10. Elapsed Time versus Relation Sizes (Faster Network)

6.4. Conclusions

In this chapter, we have described six different parallel and pipelined join algorithms. Some results from our analysis are also presented to compare those algorithms. The results of our performance evaluation reiterate the relative performance superiority of the hash based algorithms compared to sort based algorithms. Our results also show how the effects of overlaps among the different steps of an algorithm affects its relative performance. We calculate the bottlenecks in the alternative join algorithms and show that the performance of an algorithm improves by distributing the tasks across the various non-overlapping stages of the algorithm so that maximum overlap and equitable resource utilization are achieved. Our results show that intercluster communication bandwidth is typically a bottleneck and thus the algorithm or system configuration which reduces intercluster data transfer is preferred. The replicated versions of the algorithms perform typically better than their non-replicated counterparts because of the reduced intercluster data transfer.

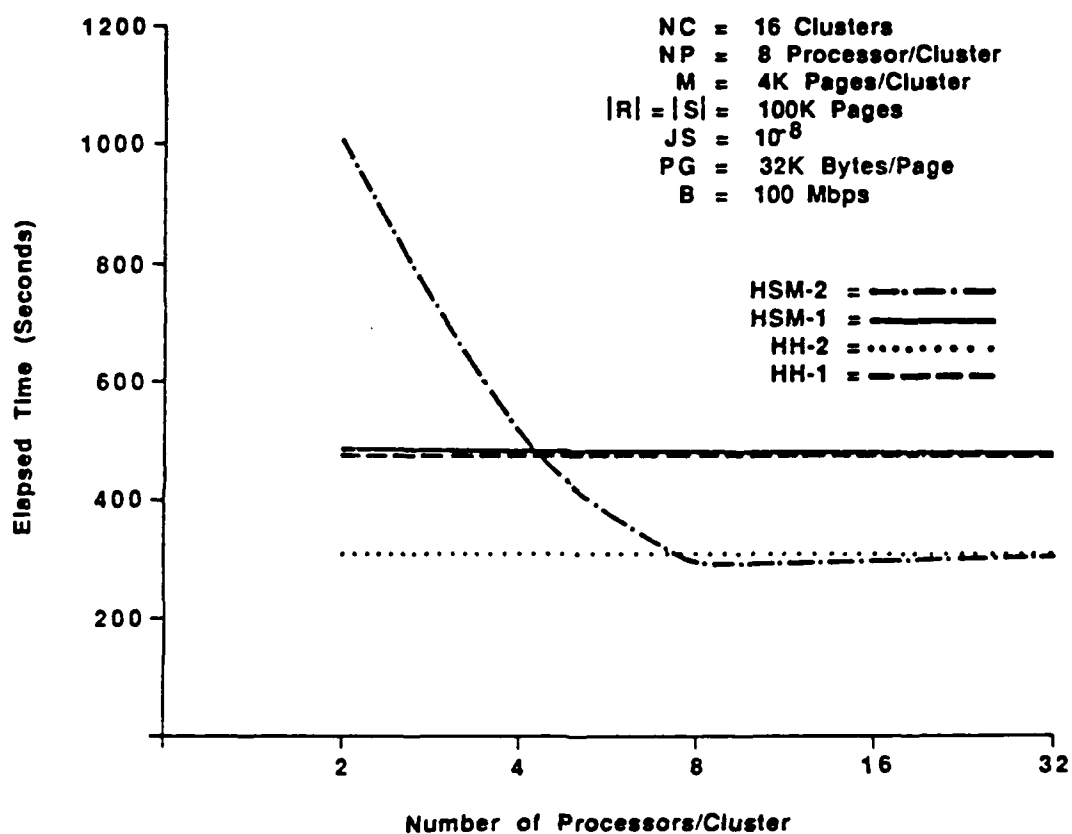From this study, we can conclude some basic conclusions regarding the parallel processing of join operations in the multiprocessor environment.

(1) The different performance shown by the algorithms studied indicates that it is important to choose appropriate algorithms for a particular join operation with a given system configuration. Furthermore, with a given system and relations to be joined, the query optimizer has to carefully determine the number of cluster, the number of disks and the number of processors which will be used in the join. Generally speaking, the hash-based algorithms outperform the sort-merge algorithms if the output tuples are not required in the sorted order. However, in the case that the source relations are already sorted, or the applications require the output tuples are sorted on the join attributes, the sort-merge algorithms may be advantageous. One possibility which is not mentioned is to use an order-preserving hash function in the hash-based sort-merge algorithms. The sorted order is maintained between different buckets and the final output tuples can thus be in the desired sorted order. The use of an order-preserving hash function should not introduce heavy extra cost.

(2) In multiprocessor-multidisk systems, high parallelism can be achieved by dividing the total processing task among processors and disks and executing the subtasks concurrently. However, in some algorithms, such as the sort-merge algorithms evaluated in this study, the parallel processing becomes difficult for some steps (final merge, for example). The increase of the number of processes cannot speed up the processing. On the other hand, the hash-based algorithms are naturally parallelizable. Both the partitioning and joining phase can be concurrently executed by all participating processors. This is the main reason that explains why the hash-based algorithms outperform the sort-merge algorithms with regard to the elapsed time.

(3) Among three major system resources, CPU, disk and communication network, CPU seems not the bottleneck of the processing pipeline in general (only in some steps of the sort-merge joins as mentioned above). For hash-based algorithms a

small number of processors at each cluster is enough to provide the necessary processing power. On the other hand, disk I/O can be the bottleneck of the pipeline, although we intentionally used large page size (32K) and very high disk-memory tranfer rate in our study. One possible approach is to increase the number of disks in each cluster. This multi-disk system can efficiently remove the bottleneck caused by slow disk I/O. However, the number of disks that can be attached to one cluster must be limited by the complexity of control.

For joins with small or moderate size relations, communications cost should not be a dominant factor in local area networks [Lu85], there is, however, still the possibility that the communications line become bottleneck when the large amount of data has to be transferred in a very large system through the communications line. This is especially true for the algorithms where a large amount of data transfer is required (such as HSM1 and HH1).

(4) One key point in the design of a parallel processing algorithm is to achieve maximum overlap among operations requiring different resources in order to increase the parallelism and reduce the effect of some resource which is the bottleneck of the pipeline. For example, in the hash-based algorithms, the remotely processed tuples can be transferred either during partitioning or right before their use in the joining phase. The total communication cost is the same in these two schemes, while their overlapping with disk I/O is different. In the first scheme, all communication occurs while the relations are partitioned. The second scheme distributes the communications cost; each relatively small amount of data transfer overlaps with disk I/O and CPU processing iin joining phases. Which scheme is better will depend on the relative speed of disk I/O and data transfer over the network. This example reminds us that parallelism between different type of resources can be further increased by tuning the processing steps carefully for each algorithm. Furthermore, the precise analytical analysis of such parallel processing algorithms is very difficult. Some simulation or tests in real systems would be useful.

Since the system configuration, that is, the number of clusters, the number of processors, the number of disks, and the size of memory used in a join operation affects the performance along with the relation size and selectivities, query optimization in this multiprocessor environment could be more complicated, and also more important. It might be a useful practice to more thoroughly investigate the relative behavior of different algorithms with regard to the parameters and derive some heuristics which can be used in the query processing process for such a data flow database machine.

CHAPTER 7

Fault Tolerance in Very Large Data Base Systems

A system for very large databases will require a high degree of parallelism and, therefore, will involve a large number of components. In such a system, many components can fail, affecting the performance of active components and the availability of the system. Designing such a system with a desired degree of fault tolerance is difficult and requires qualitative and quantitative considerations of factors affecting system fault tolerance.

In this investigation, we study the effect of fault tolerance techniques and system design on system availability. Specifically, we attempt to answer the following questions: What are the main parameters that affect fault tolerance of a very large database system? How do you evaluate their effect on fault tolerance? How important are various fault tolerance techniques? What are the trade-offs that should be considered when designing a very large database system with a desired degree of fault tolerance? A generic multiprocessor architecture is used that can be configured in different ways to study the effect of system architectures. Important parameters studied are different system architectures and hardware fault tolerance techniques, mean time to failure of basic components, database size and distribution, interconnect capacity, etc. Quantitative analysis compares the relative effect of different parameter values. Results show that the effect of different parameter values on system availability can be very significant. System architecture, use of hardware fault tolerance (particularly mirroring) and data storage methods emerge as very important parameters under the control of a system designer.

7.1. Introduction

A very large database is usually heavily used, and many users and applications depend on it. Downtime or unavailability of such a system is expensive and affects critical applications dependent on it. A system can become unavailable because of the faults in one or more of its components. To increase its availability, the system must tolerate components faults. It is also desirable to contain or tolerate faults, because recovery after a failure that affects the data can be very costly if the database has to be reconstructed.

A system that gives good performance and manages a very large database has many components and a high degree of parallelism. In such a system, even though any individual component may be fairly reliable, the probability that one of the components will fail becomes much higher than the probability that an individual component will fail. Since many components need to cooperate in a parallel processing system, achieving fault tolerance is more difficult. However, because the system is large and there is

more redundancy, there are more opportunities to achieve fault tolerance.

The topic of fault tolerance in computers and database system has received a fair degree of attention. Articles by Kim [Kim84] and Siewiorek [Siew84] discuss architectures of fault-tolerant computers and fault tolerance techniques used in many commercial and prototype systems. Some of the publications that discuss fault tolerance techniques used in commercial and prototype computers are [Siew78, Borr81, Kast83, Borr84, Bern85] and [Gray86]. Recently, work continues on larger systems, such as Teradata DBC1012 [Deck86], but not much has been published on the fault tolerance techniques used in such systems or on evaluation of such techniques. While it seems that many of the techniques developed for smaller systems are applicable to larger systems, designing fault tolerance for large systems is more difficult because they have many more components that can fail and there are more alternatives to provide fault tolerance. [Koon86] and [Shet87] evaluate fault tolerant algorithms in a relatively small (~ 10 nodes) and loosely coupled distributed systems. However, there is no published research on evaluating fault tolerance in a tightly coupled multiprocessor system for large databases (> 100 gigabytes).

In this chapter, we study the effects of various system parameters and fault tolerance techniques in a system for very large databases. We use a generic multiprocessor architecture which, by choosing different system parameters can represent a range of system architectures, from a loosely coupled multiprocessor with non-shared memory and partitioned database to a tightly coupled multiprocessor with shared database and shared memory. We study the effects of various architectures, fault tolerance techniques, mean time to failure of important components, database size and distribution, interconnect capacity, etc. on the availability of a system. The results show that the choice of different system architectures and some parameter values significantly affect availability. Because fault tolerance is obtained at the cost of additional system resources (in terms of number of redundant components and system time and resources required to maintain redundant information), a designer's task is to minimize such cost to obtain the desired level of fault tolerance. The results can help a designer to understand important trade-offs and choose an appropriate system architecture and fault tolerance techniques.

This chapter is organized as follows. Section 7.2 describes the generic system architecture we evaluate. Section 7.3 describes basic concepts and terminology as well as various fault tolerance techniques. Section 7.4 defines two availability measures, one of which we evaluate in detail to measure fault tolerance. Section 7.5 describes the quantitative analysis and the results. Section 7.6 discusses our conclusions.

7.2. System Description

Achieving good performance in a very large database system (VLDBS) requires a large amount of computing power. Given the limitation of the computing power of a single component, we use a high degree of parallelism. A VLDBS has these characteristics:

1. It has many major components such as processors, disks and memory.

2. The relations are very large and distributed over many components for storage and parallel-processing. Because of the very large size of the database, the amount of replication possible is limited.

3. An update transaction will change ma..y data items. This means that reintegration of failed components can take longer.

We assume a generic multiprocessor architecture for a VLDBS, as shown in Figure 7.1. This architecture consists of a set of "clusters" linked by an interconnect. Each cluster consists of a set of processors, a shared memory bank addressable by all processors in the cluster, and a set of disk storage units and associated controllers. Each cluster has its own power supply. All the components within a cluster (processors, disks/controllers, memory and power supply) are connected by an intracluster bus. The size of a VLDBS is defined by the architectural parameters given in Table 7.1. A *component unit* (e.g., a processor unit or disk unit) consists of one or more components. A component unit consisting of k components is called a *k-redundant* unit in which *k-1* components are redundant components used to increase fault tolerance. However, all active components in a unit perform the same function, so the redundancy does not add to the computing power (in the case of the processor unit) or storage capability (in the case of the disk unit or memory unit). Our architecture has one power supply unit, one memory unit, and an intracluster bus per cluster.
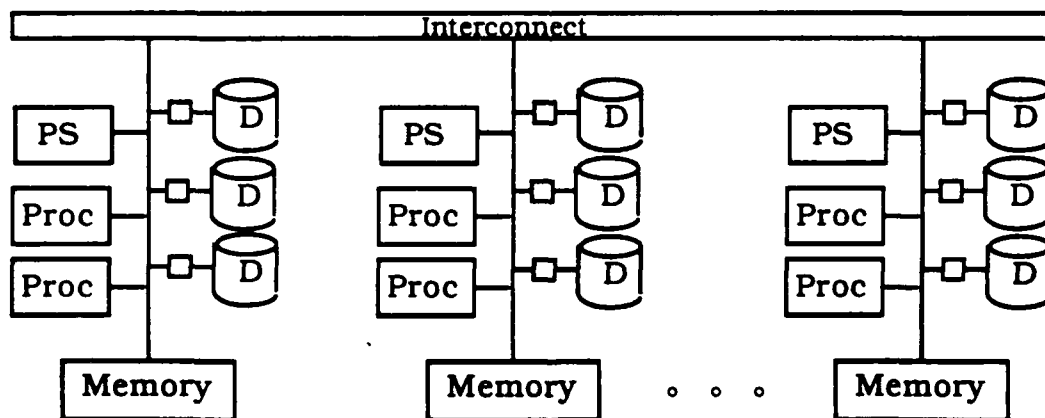

Figure 7.1. Architecture of a VLDBS

NC	Number of clusters
NP	Number of processor units per cluster
ND	Number of disk units per cluster

Table 7.1: Architectural Parameters

7.3. Fault Tolerance Techniques

In this section, we discuss basic concepts and principles of fault tolerance (section 7.3.1), hardware fault tolerance techniques (section 7.3.2), software fault tolerance techniques (section 7.3.3), and data storage methods (section 7.3.4).

7.3.1. Basic Concepts

A component or a subsystem that functions according to its specifications is called *active.* Three terms are relevant to the operation of the components and the system: fault, error, and failure [Siew82, Aviz84]. A *fault* is a defect in a component of the system. *Faults* result in *errors*, which are undesired or invalid component states. Errors may result in *failures*, which mean loss of the service expected from the component (or the system of which the component is a part). Errors may be transient, intermittent, or permanent. The first two are also referred to as soft errors, while permanent errors are referred to as hard errors. Siewiorek and Swarz [Siew82] estimate that soft errors account for more than 90% of all faults. Failures from soft errors are called *soft failures*, while those from hard errors are called *hard failures*. The fault tolerance techniques should protect the system against soft as well as hard failures.

After the failure of a component (or subsystem), the system may take a corrective action called *failure recovery.* This may involve *reconfiguring* the system to isolate the failed component and to reorganize the system so that it can be restarted without the failed component. Once the failed component is repaired, it is *reintegrated* with the system. In a multiple-component system the failure of a single component that affects no other component during the recovery is called a *single failure.* Simultaneous failure of more than one component, or a failure of a component while the system is recovering from a previous component failure are called *multiple failures* (or *double failures*, if two components fail as defined). Normally, the probability of a multiple failure is very low. After a failure of one or more components, a system that can continue to operate at lower efficiency corresponding to the loss of power associated with the failed components is called a *gracefully degradable* system.

The following criteria are desirable in a highly available database system [Kim84]:

1. The system must guarantee database consistency by providing transaction processing with concurrency control, distributed commit, and recovery techniques.

2. The system must support *automatic* recovery when failure occurs. A backup process should automatically take over when the primary process fails.

3. The system must survive at least a single failure (and possibly multiple failures) of major system components, including the processor, disk drives, memory, and inter-process communication medium.

4. The system should support on-line reintegration of failed components when they are repaired or replaced.

5. Other features, such as the automatic restart of transactions affected by failures and the rerouting of messages to bypass failed communication route, are desired.

Two basic principles are used to achieve fault tolerance:

1. *Modularity* — By modularizing the system, the modules become units of failure and replacement.

2. *Redundancy* — By having a redundant module (a hardware or a software resource), the primary component can be replaced with the redundant component if it fails.

The principle of modularity must be taken into account while designing the system. The multiprocessor system evaluated is made modular by providing various hardware fault tolerence techniques to tolerate component failures, and by designing a clustered architecture so that the system can be partially available using reconfiguration when some clusters fail.

Redundancy is a basic property required for all fault tolerance techniques. It is used to provide information needed to negate the effect of failures [Siew84]. Redundancy can be obtained by having extra components, or *physical redundancy*, and by having extra time, or *temporal redundancy*. Physical redundancy is used as hot stand-bys (or backups), as checkers that mask faults via voting, and to reconfigure the system around faulty components. Temporal redundancy is used for retrying operations to recover from transient or soft errors.

The basic physical redundancy technique used to increase the availability of each of the components include duplication (2-redundancy), triplication (3-redundancy) and voting, and k-redundancy. We will briefly discuss the first two techniques.

In the duplication technique, two identical components (processors, disks, or memory) are used in parallel. Duplication can be used in two ways. One technique is to use the second component as a hot standby. In this technique, if either component fails, the other component takes over. This technique is especially useful to provide fault tolerance against hard failures. Examples of this type of duplication are disk mirroring and memory mirroring, which are discussed in the next subsection. In the second duplication technique, the two components are driven by the same input, and their outputs are compared. If the outputs are not the same (i.e., they vote differently), the operation is retried. This technique is especially useful for providing fault tolerance against soft failures and for detecting potential hard failures. An example of this type of duplication is processor pairing, also discussed in the next subsection.

The triplication component is usually used for the triple modular redundancy (TMR) technique. Here three components are used in parallel. Outputs of all components are compared, and the component unit continues to function as long as at least two outputs match (i.e., at least two votes are received). The technique is most useful

for masking errors by voting. Thus. it is helpful for soft failures but not hard failures.

7.3.2. Hardware Fault Tolerance

In this subsection. we will examine the fault tolerance techniques for the major system components. These include processors. disks, memory, power supply, intracluster bus, and interconnect.

7.3.2.1. Processor Pairing

The reason for duplication of processors is to check for soft errors and increase fault tolerance by eliminating them. Like other components. a processor experiences two types of failures. hard and soft. The mean time to failure (MTTF) for hard failures for microprocessors that are used in highly parallel systems is estimated to be 100 Khrs. The rate of soft failures is estimated to be an order of magnitude higher: hence a soft failure MTTF is estimated at 10 Khrs. With a high degree of parallelism. this failure rate can have a disastrous effect.

In processor pairing. the same input and clock drive two processors. but the system uses output of only the primary processor. Thus. the second processor does not add computing power to the system. However. the output of the secondary processor is used to compare with that of the primary processor. and differing output signals an error. An error triggers the instruction retry. Thus the processor pairing attempts to mask soft failures at the instruction level. It also helps to prevent error propogation. If the error persists after several retries. it is identified as a hard error. and both the processors turn themselves off after raising an interrupt. Microprocessor-based fault-tolerant system vendors. such as Stratus [Kast83] and Sequoia [Bern85] have taken similar approaches. Gray [Gray86] discusses several approaches for designing processor pairs. It is also possible to use the processor pairs to tolerate hard failures by using a different scheme. but we will handle hard processor failures by using redundancy in processor units in a cluster and keep the technique for processor pairing simple.

7.3.2.2. Disk Mirroring

In most systems. disks are the biggest reliability problem, because they are complex systems containing a fair number of mechanical components. Disks contain circuitry to detect and tolerate soft failures; so the VLDBS does not deal with soft failures within a disk. The hard failure MTTF of a disk is estimated to be 10K hours. Disk mirroring. also called duplexing, is used to increase this MTTF.

The technique used for mirroring disks is generically called *reconfigurable duplication* [Siew82]. A disk unit in this duplication technique consists of two disks. Both disks receive the same input through independent paths and controllers and store on respective media. Thus. both disks are in *sync*. When both disks are active. one acts as a primary and the other as a hot standby. If either of the disks fails. then the other disk takes over. The system can run with only one disk operating correctly. When a failed disk is repaired. its contents are updated so that both disks again contain the same information. A repair may mean replacement. Tandem uses the disk mirroring

technique in its fault tolerant systems.

7.3.2.3. Techniques for Memory

The shared memory in our architecture is a random access memory. As with other components. the memory unit suffers hard failures as well as soft failures. The soft failures can be classified as *intermittent* and *transient* failures and are much more frequent than hard failures. Intermittent failures occur under conditions such as system overload. while transient failures are due to external conditions such as voltage fluctuations.

Techniques to tolerate soft failures include *error detection and correction techniques* such as parity codes. In case of hard failures, the failed chip or a memory bank containing the failed chip is replaced. A hard failure can be tolerated by a *reconfiguration*. which isolates the failed chip. Duplication is another fault tolerance technique sometimes used. The duplication can be used to detect soft failures as in processor pairs. or it can be used for *mirroring* as in mirrored disks to tolerate hard failures.

7.3.2.4. Techniques for Power Supply

Two components of power need to be considered: power that comes to the system as a whole. called *power source*. and the power that is supplied to each of the clusters. called *power supply*. The power coming from the source is usually conditioned. To improve the fault tolerance. we expect that an uninterrupted power supply (UPS) will be used. A UPS uses a battery that immediately takes over if the normal power source fails. When the system draws power from the battery, it may be run at reduced capacity to save power. Next. we expect that each cluster will use at least one power supply. By having an independent power supply for each cluster. the failure of a power supply can directly affect only one cluster. If the cluster size is big. more than one power supply per cluster is recommended.

7.3.2.5. Techniques for Intracluster Bus

This component connects all the processors. memory, and disks within a cluster. We expect it to be a short (within one cabinet) and very high speed bus. It will occasionally suffer from soft failures. but we assume that the low level protocols and the hardware will tolerate them. A hard failure of this component is extremely infrequent. but if it occurs. it can be modeled as the failure of one or more components that are affected.

7.3.2.6. Techniques for interconnect

The type and frequency of failures in an interconnect that connects all the clusters in the system will depend on the type of interconnect (e.g.. ring. hypercube. bus) and its topology. For the sake of simplicity. we will assume a generic interconnect. An interconnect may suffer a soft link failure, which may corrupt or lose data. These errors can be detected by check codes and time outs. Most of these errors are recoverable by the protocol (e.g.. retransmission) or taken care of by error correction codes. If the soft

error is not recoverable, the transaction is aborted. Hard failures can occur because of the physical breakup of a link or failure of the cluster's connection to the interconnect. The basic technique to handle hard failures is duplication in the interconnect (e.g., double ring). Each cluster can have independent paths to two interconnects to handle hard failures.

7.3.3. Software Fault Tolerance

Basic software fault tolerance techniques can be divided into three components: (1) transaction fault tolerance for failures that affect transaction execution, (2) system software fault tolerance for failures that affect system software (e.g., operating system and communication software), (3) and data fault tolerance for failures that affect data availability.

A soft failure may occur in the system because of transient or intermittent failures in its hardware components or software. These errors may corrupt the system software as well as fail transactions. For example, a transient error in the shared memory may corrupt the system software. Alternately, such a failure may corrupt a transaction's work space. The techniques discussed in transaction fault tolerance (section 7.3.3.1) and system software fault tolerance (section 7.3.3.2) aim to tolerate such soft failures. Section 7.3.3.3 discusses tolerating hard failures that have a more serious impact on system functioning and require more elaborate recovery schemes such as the one discussed in section 7.3.3.4.

7.3.3.1. Transaction Fault Tolerance

In this subsection we discuss transaction failure and recovery.

A transaction is a set of operations performed on the database. It should have four properties (recognized as ACID properties), [Haed83, Gray86], atomicity, consistency, integrity, and durability. Two of these properties are of main interest to us:

- *Atomicity* — Either all or none of the operations in the transaction should be performed. The transaction commits if all the operations are performed; otherwise it aborts. This property should be guaranteed even in case of failures (this property is sometimes called failure atomicity).

- *Durability* — Once a transaction commits, all of its effects must be preserved, even if there are failures.

Transaction fault tolerance techniques aid the concurrency control mechanism to guarantee the four properties even if there are failures. In particular, a two-phase commit protocol is used in a distributed database environment to guarantee that either all or none of the copies of data are updated when a transaction executes. If there is a failure that prevents the transaction from completing all its operations, none is performed and the transaction is aborted.

A transaction may fail because of a hardware (component) fault or a software fault in the system. Examples of the faults are incorrect execution of the transaction (e.g., a divide by zero); failure of a component on which the transaction was executing (e.g., a

processor failure); and a transient error in the transaction's work space in memory. The approach to handling these failures is simple. In two-phase commit, a temporary copy of data resulting from updates is stored on the disk (called safe storage). If a failure affects the transaction, the transaction is aborted. If the failure has resulted from a soft fault such as transient error in the transaction work space, the transaction is restarted. If the failure has resulted from a hard fault but is recovered (e.g., by reconfiguration as discussed in section 7.3.3.4), the transaction is restarted. For example, if the processor executing the transaction fails, the transaction is aborted and restarted by assigning it to another processor in the same cluster. If the disk in a cluster fails, the cluster fails. In this case, all transactions in the failed cluster are aborted. After the recovery, which is discussed in the next subsection, the transactions may be restarted.

When a failure affecting a transaction occurs, it is important to efficiently abort the transaction and bring the database and the system to a consistent state. There are two basic models of transaction execution that lead to different techniques for transaction abort. One model is called an *UNDO* model, in which the transaction writes directly in the database while executing ("write in place"). When the transaction is aborted, the transaction mechanism performs UNDO operations for all the operations performed by the aborted transaction. In the other model, called the *work space* model, a transaction writes in a work space (also called differential files) until it is committed. Upon commitment, it writes the changes stored in the temporary database into the permanent database. If a transaction is aborted, the work space is simply discarded. We prefer the latter model.

To aid in transaction restart and system reconfiguration, at least two copies of transaction information are maintained in the system (see section 7.3.3.2).

7.3.3.2. System Software Fault Tolerance

In this subsection, we discuss how to tolerate failures that affect the system software. We will only discuss the soft failures that corrupt the system software because the hard failures are handled as in the data fault tolerance.

We considered transaction failures in the previous subsection. We use the traditional way of reinitialization (or reboot) to recover from the failures that corrupt the system software. The reinitialization may be limited to the cluster in which corruption occurs. Many systems today have multiple levels of system reboot procedures, often referring to a more extensive reboot as a *cold reboot* and to a less extensive or limited reboot as a *warm reboot*. Proper care in developing system software, especially recovery techniques, will limit software reinitialization to a warm reboot.

7.3.3.3. Data Fault Tolerance

Availability of the database greatly depends on how the data is distributed and duplicated. The data fault tolerance is attained by proper placement of data. Our primary method is to have **two copies of each data item**. To do this, we first horizontally fragment a relation and assign different horizontal fragments to different clusters. A fragment assigned to a cluster is further vertically fragmented, and different vertical

fragments are assigned to different disk units in a cluster. In this scheme, multiple disk units within a cluster are used to improve response time but not fault tolerance. Later we briefly discuss a data quadruplication scheme in which multiple units within a cluster are also used to improve fault tolerance.

Maintaining two copies and multiple fragments of each copy can also simplify and improve the retrieval performance of the system, but we will not address that issue here. We use three rules in our primary scheme to distribute the database:

1. The database is partitioned across the clusters. Thus the failure of a cluster will make only the fragment(s) assigned to it unavailable.

2. Each partition assigned to a cluster is uniformly distributed over all the disk units in that cluster. It may be noted that the purpose of having multiple disks in a cluster is to increase the parallelism and not the fault tolerance.

3. The two copies of data (fragment in our scheme) cannot reside in the same cluster for the sake of fault tolerance.

Fragment Allocation Scheme

Now let us discuss how fragments are allocated to different clusters. It is important to allocate the fragments of each of the two copies to the clusters in an intelligent way because it affects the fault tolerance. For example, consider dividing the database into four fragments so that two copies of each fragment are distributed over four clusters. Two of the possible data distribution schemes are shown in Figures 7.2 and Figure 7.3. In both schemes, the database will remain available if there is a single failure of clusters because one copy of every fragment will still be accessible. In the first scheme the database will be available in two out of a possible six double failures of clusters. In the second scheme the database will be available in four out of a possible six double failures of clusters. Thus, the data placement in the second case is preferable. Because the probability of double failures is negligible, we will not address recovery of double (or multiple) failures of clusters.

Now let us look at how the system can be reconfigured in terms of data placement after one or more single failures of clusters. Consider the failure of cluster 1 when the scheme shown in Figure 7.2 is used. In this case, one copy each of fragment F1 and fragment F4 will be lost, and the other three clusters have only one copy of fragments F1 and F4. Although it is possible to continue processing the transactions, it is not desirable to do so, because it will not allow the system to be gracefully degradable. Failure of one more cluster may leave the system with no copy of a fragment. For example, failure of cluster 2 would mean that no copy of fragment F1 will be available, and so the system can no longer process transactions.

The solution is to create a second copy of the fragments that are unavailable because of a cluster failure. Following the data placement rules discussed above, the system of three working clusters can be reconfigured as shown in Figure 7.4. Following the same arguments, if cluster 3 were to fail after reconfiguration before cluster 1 failed (as shown in Figure 7.4) but before cluster 1 is reintegrated, the system can be

Double failures in which database will be unavailable

Double failures in which database will be available

FI First copy of fragment i

FI Second copy of fragment i
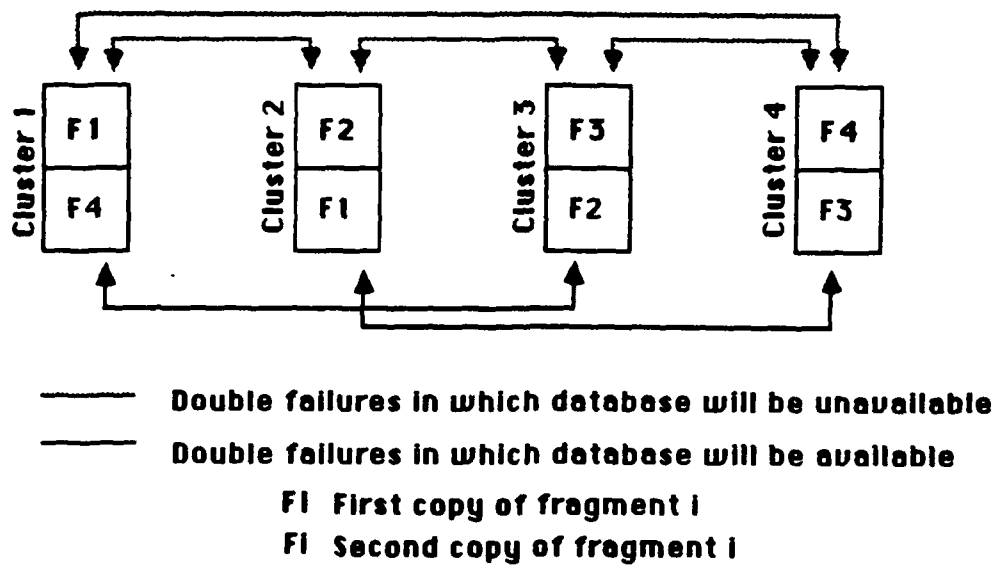
Figure 7.2. Data Placement Scheme 1

Double failures in which database will be unavailable

Double failures in which database will be available

FI First copy of fragment i

FI Second copy of fragment i
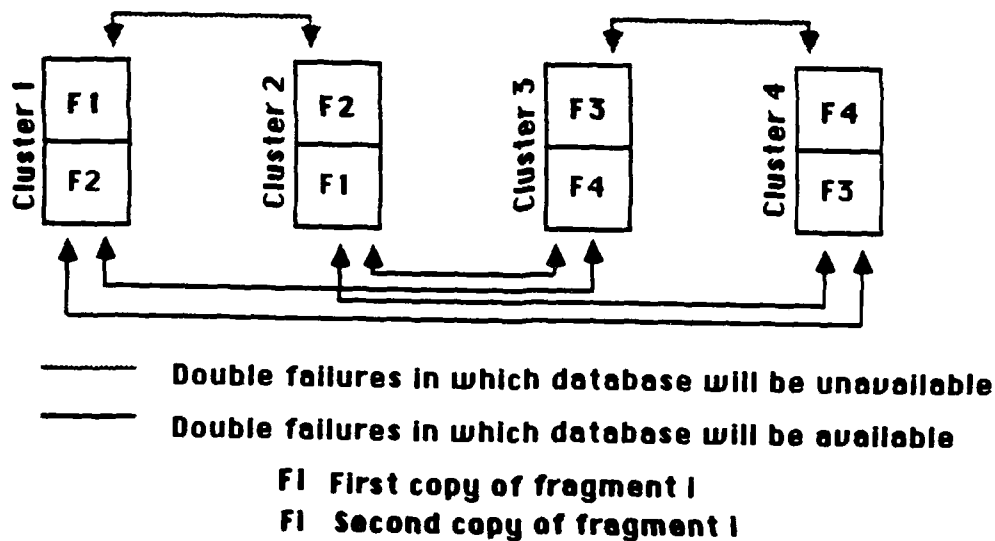
Figure 7.3. Data Placement Scheme 2

reconfigured as shown in Figure 7.5. Because at least two copies of data must be kept in the working system and the two copies cannot be stored on the same cluster, the system can tolerate faults until two clusters are left in the working system.

After a failed cluster is repaired, it is reintegrated into the system. This involves replacing (or updating) the copies of the fragments on the repaired cluster with the up-to-date copy in the rest of the working system. For example, if cluster 1 was repaired
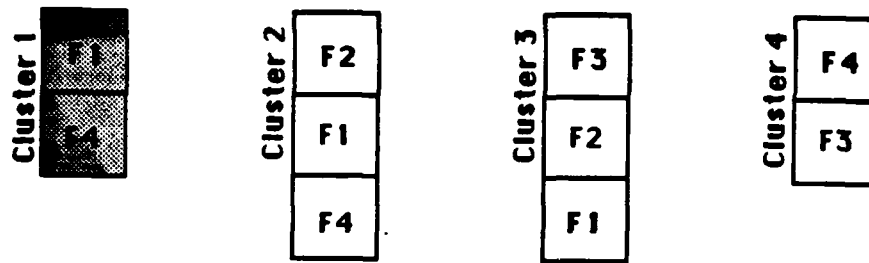
Figure 7.4. Data Placement After Cluster 1 Fails
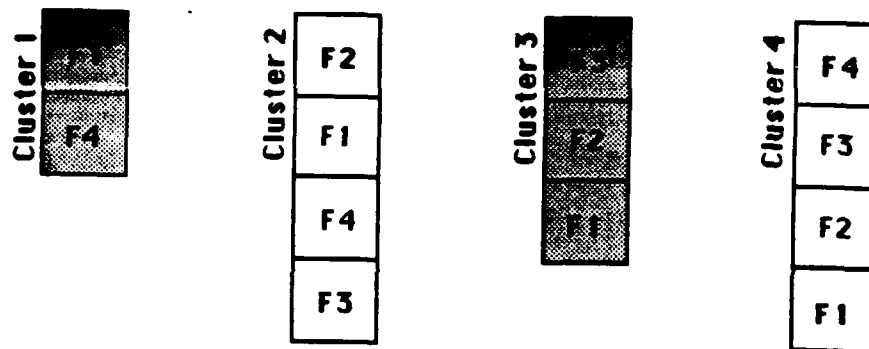
Figure 7.5. Data Placement After Cluster 3 Also Fails

after the situation shown in Figure 7.5, fragments F1 and F4 will be updated using either of the two up-to-date copies on clusters 2 and 4, fragment F1 will be deleted from cluster 4, and fragment F4 will be deleted from cluster 2. The process of replacing (or updating) can occur in the background for the active clusters (clusters 2 and 4 in this case) for most of the time. However, just before cluster 1 is made active, there may be a brief pause in the system to allow for the reallocating transaction and for updating the global data allocation directories. We leave out some details of cluster reintegration.

The above scheme is based on data duplication. If the fault tolerance achieved by maintaining two copies does not meet the requirement, the amount of data replication can be increased. However, this can be done only at the cost of performance since the more copies of a data item, the more overhead of storage and update synchronization. We will briefly discuss a **data quadruplication** strategy, which may improve the fault

tolerance substantially.

The quadruplication strategy maintains two copies of data within one cluster and two additional copies in another. The two copies within the clusters can be distributed in two ways, giving two different quadruplication schemes. In the first scheme, a disk unit consists of mirrored disks. In this case, logically, there are only two copies of every fragment, but physically there are four copies of fragments. The hardware keeps two identical copies on every mirrored disk. Since system software uses the addressing, the data update algorithm knows about only two copies. In the second scheme, the same strategy of fragmentation used across the clusters is also used now for the two copies across the disk units within a cluster. In this case, the data update algorithm knows about the four copies and has to send update commands to all four copies separately. Thus, in this scheme the cluster cannot fail if a disk unit goes down in a cluster (just as the system will not go down if one cluster goes down). We feel that the first scheme is simpler and has less overhead when there are no failures. Overheads for the two schemes when failures occur remain to be studied.

7.3.3.4. Recovery From Hard Failures

A hard failure is usually more serious, requiring more expensive methods to tolerate it. There are two types of hard failures with respect to the type of recovery required — one in which the availability of the data is not affected and the other in which the data becomes unavailable. The latter type is like a media failure in traditional systems and requires more an elaborate recovery process. An example of the first type of failure is a hard failure of a processor unit (a single processor or a processor pair). An example of a second type of failure is the failure of the shared memory or the failure of a disk unit (a disk or mirrored disks), both of which lead to the cluster failure in our primary scheme of data distribution.

A component suffering a hard failure remains out of commission for a relatively long time, and we cannot let the system be unavailable for that time. The process we use to recover from hard failures is called reconfiguration. In this process, the following actions are taken: The failed component is logically isolated from the system so that it can be either repaired or replaced. Secondly, in some cases, spare replacement components can readily replace the failed components. If a replacement is made, the system can function at the original capacity once the reconfiguration is complete. If a replacement is not made, the system will function at a lower capacity proportional to that loss because of the component failure. Third, the functions of the failed component are redistributed to the functioning components. If a part of data becomes unavailable, then that data is made available by making a copy of it.

Next let us look at recovery when a cluster fails and data availability is also affected. Let us first discuss the important design features that help in the recovery.

Adjacency — There are many ways in which a cluster failure can be detected. For the sake of brevity, we will discuss only one possible scheme. We arrange all the clusters in a logical *unidirectional ring*. Periodically, each cluster sends an "I am alive" message to the cluster to its left. If a cluster does not receive this message within a predetermined

time, it will communicate with the node to its right to ascertain that it is dead.

Duplicate Transaction Information — It is necessary to keep at least two copies of information on the active transactions in the system. To store this information, each cluster maintains a *transaction table*. Information about each active transaction can be maintained in two transaction tables, one in the transaction table in the cluster of transaction origin, and other in the transaction table in the cluster to its left.

Global Data Allocation Directory — A global data allocation directory is a bidirectional list of *data fragment → cluster*.

Duplicate Lock Information — Each cluster maintains a *lock table*. When a transaction accesses a data item, a lock is set on both copies of the data item, which are in different clusters. The respective lock tables maintain this information.

Component	Failure Type	Failure Handling Method
Processor	Soft Failure	Processor Pairing
	Hard Failure	Transaction Recovery, or
		Software Reinitialization, or
		System Reconfiguration (in extreme cases)
Disk	Soft Failure	Error Correction Codes and
		Hardware Techniques
	Hard Failure	Disc Mirroring, or
		System Reconfiguration
Memory	Soft Failure	Parity and Error Correction Codes, or
		Transaction Recovery, or
		Software Reintialization
	Partial Failure	Memory Reorganization
	Hard Failure	Memory Mirroring, or
		System Reconfiguration
Power Supply	Soft Failure	Hardware Techniques
	Hard Failure	System Reconfiguration
Intracluster Bus	Soft Failure	Hardware Techniques
	Hard Failure	Treated as failure of associated component
Interconnect	Soft Failure	Protocol/Retries, and
		Error Correction Codes
	Hard Failure	Redundant Paths

Table 7.2: Failure Tolerance Techniques As Applied To Various Failures

In certain failures, it is necessary to isolate the failed component and continue the system operation. When a component is isolated, access to some data may be lost until the failed component is repaired. Our data storage scheme dictates that at least two copies of any data must be available to allow tolerating any additional failures and graceful system degradation. The reconfiguration scheme depends on system hardware

and software configuration, including the data storage scheme. We will now discuss the system recovery scheme used to tolerate some of the hard failures that result in cluster failure. We will assume the data duplication scheme defined earlier.

1. **Detect Cluster Failure.** *Some of the hard failures result in the cluster failure.* A failure of a cluster is detected using the adjacency feature described above. For simplicity, we will discuss a centralized reconfiguration algorithm in which the adjacent cluster that detects the cluster failure coordinates the reconfiguration.

2. **Prepare for reconfiguration.** Upon detection of a cluster failure, the system enters a *pause state*. During this step, all the committed transactions are completed. All the transactions, which are active but not yet committed, are aborted. This step requires accessing and updating appropriate transaction tables and lock tables. The aborted transactions are restarted when the system leaves the pause state. The system remains in the pause state until the reconfiguration is completed. While the system is in the pause state, newly arriving transactions are queued behind the transactions to be restarted.

3. **Find what is missing and locate the redundant copy.** The coordinator accesses its global data allocation directory to find out the data (fragments) that was (were) stored on the failed cluster and to locate the respective redundant data (fragments). We call the cluster with redundant data source clusters.

4. **Decide the Destination Clusters.** These are the clusters where the second copy of the data has to be created from the copy of data on the source clusters.

5. **Send the Data to the Destination Clusters.** The coordinator instructs the source clusters to send the relevant data to destination clusters. A destination cluster decides how to store the data on its disks.

6. **Update the Global Data Directories.** The coordinator sends appropriate information to all the active clusters to update their respective global directories.

7. **Restart.** The coordinator signals *all O.K.* to all the clusters. The system then leaves the pause state and start processing transactions in its queues.

After the repair of one or more failed components, a cluster may be ready to be reintegrated with the rest of the active system. The reintegration process requires that the repaired cluster is updated with respects to the data. Most of this process can be performed in the background. For the sake of brevity, we do not provide further details.

7.4. Measuring Fault Tolerance

The parameter used most widely to characterize the fault tolerance of a system is its availability. To measure the availability of a system and to devise fault tolerance techniques to achieve the desired level of system fault tolerance, we should also be able to measure availability of each of the system components.

7.4.1. Component Availability

The term fault tolerance reflects the ability of a system or component to tolerate a fault and remain active. As discussed earlier, availability of a component or a system reflects its fault tolerance. Two parameters of a component are relevant in calculating a component's availability: MTTF, mean time to failure, and MTTR, mean time to repair. MTTF refers to the average time that elapses between two consecutive failures of a component. MTTR refers to the average time needed to detect and repair a failed component. The availability of a component can simply be given by the ratio,

$$Availability = \frac{MTTF}{MTTF + MTTR} \tag{7.1}$$

Estimates of the MTTF and MTTR of each component of a system should be known before system availability can be estimated. Table 7.3 gives the estimates of MTTF for the components that we will use in our analysis. These estimates have been derived from the literature.

Component MTTF Estimates	
Processor	100K hours
Memory (64 MB RAM)	100K hours
Disc	10K hours
Communication	100K hours
Power Supply	100K hours

Table 7.3: Component MTTF Estimates

7.4.2. System Availability

Availability of a fault tolerant database system can be defined in one of two ways. We call them strong availability criteria [Smit86] and weak availability criteria [Shet87].

Strong Availability: Strong availability of a database system is given by the percent of the time the *entire* database is available for access by authorized users, i.e.,

$$S\_Avail = \frac{time\ the\ entire\ database\ is\ available}{total\ time} \times 100 \tag{7.2}$$

Weak Availability: Weak availability of a database system is given by the percent of the time a transaction can be processed without being aborted because of a failure in the system. The system is available as far as the transaction submitted to the system can be processed without a delay or with a *reasonable* delay. An example of a reasonable delay is the recovery time for one soft failure during a transaction execution. Let:

$P(nf)$ = Probability that *no failure* occurs while a transaction executes, and
$P(sf)$ = Probability that there are no multiple failures during the transaction

execution and that the system recovers from every *single failure.* In other words, this is a probability that the system goes into a pause state at least once during a transaction execution.

Then,

$$W\_Avail = P(nf) + P(sf) \qquad (7.3)$$

There may be two reasons for using S_Avail as the availability criteria [Kim84, Smit86]. First, it is easy to quantify S_Avail as compared with W_Avail. Second, it is easier to justify that the specified fault tolerant techniques meet certain availability goals.

7.4.3. Assumptions

Our analysis is not a detailed analysis of a particular system. We make several simplifying assumptions without compromising the basic nature of the problem. Some of the important assumptions are as follows:

1. Each system component or module is fail fast, i.e., it either functions properly or stops [Schl83].

2. Each component fails and recovers independently.

3. Mean time to failure of a component is exponentially distributed with mean MTTF. Similarly, mean time to repair of a component is exponentially distributed with mean MTTR.

4. Components (particularly interconnect) are lightly loaded during recovery. To include the queuing effect in calculating $MTTR_{cluster}$, a more detailed model (e.g. [Shet85]) needs to be used.

7.5. Quantitative Analysis

In this section, we will study the availability of a VLDBS. In section 7.5.1, we study the availability of subsystems that consist of multiple non-redundant and redundant components. A VLDBS comprises the subsystems of components. In sections 7.5.2 and 7.5.3, we study the mean time to failure of a cluster and time to recover from a cluster failure, respectively. These two parameters are used to calculate system availability of a VLDBS in section 7.5.4. Each of these sections discusses basic quantitative methods to calculate output parameters using input parameters, followed by examples and evaluations using a range of input parameter values.

7.5.1. Component Availability

In this subsection, we calculate the availability of subsystems consisting of multiple nonredundant components (section 7.5.1.1) and multiple redundant components (section7.5.1.2).

7.5.1.1. Availability of Subsystems Consisting of Nonredundant Components

A system with no redundant components requires that all the components are active for the system to be active. Such a system is also called a *series system*. Let the mean time to failure of the i th component be $MTTF_i$. Then the mean time to failure of a system of n components, $MTTF_{ser-sys}$, is given by:

$$MTTF_{ser-sys} = \frac{1}{\sum_{i=1}^{n} \frac{1}{MTTF_i}} \tag{7.4}$$

If a subsystem has k identical components and if mean time to failure of each component is $MTTF$, then mean time to failure of the subsystem, $MTTF_{k-ser-sys}$, is given by reducing (equation 7.4). i.e.,

$$MTTF_{k-ser-sys} = \frac{MTTF}{k} \tag{7.5}$$

7.5.1.2. Availability of Subsystem Consisting of Redundant Components

In section 7.3.2 we discussed using redundancy to increase fault tolerance of a component. Consider a subsystem that has reconfigurable duplication as in disk mirroring. Such a subsystem consists of a pair of identical components working in parallel and performing the same task. The subsystem can perform the required task as long as at least one of the two components is active. Using a combinatorial model of system availability, we can map this subsystem to a *two module, two repairman model* (or alternatively a Markovian queue called M/M/2/2/2 queue) as follows. We note that the system can be in one of the three states (see Figure 7.6). In state 0, both components are active. In state 1, one of the component has failed, but the other is active. In state 2, both the components have failed. An arc joining two states specifies the rate at which the subsystem changes from the state at the tail of the arc to the state at the head of the arc. The two state transition rates shown over the arc are defined as follows (assume mean time to failure and mean time to repair of each component to be $MTTF$ and $MTTR$, respectively).
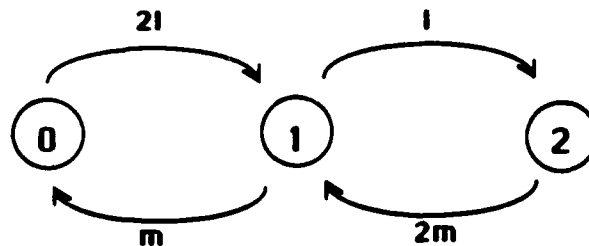

Figure 7.6. Model of Reconfigurable Duplication

$$failure\ rate\ =\ \lambda\ =\ \frac{1}{MTTF}$$

$$repair\ rate\ =\ \mu\ =\ \frac{1}{MTTR}$$

Now availability of this subsystem. A_{pair} can be calculated from the probability that the system will be either in state 0 or state 1 and can be given as follows [Siew82], page 280:

$$A_{pair} = \frac{2\lambda\mu + \mu^2}{\lambda^2 + 2\lambda\mu + \mu^2} \tag{7.6}$$

Repair time for this subsystem, $MTTR_{pair}$, is given by $MTTR/2$, because the subsystem is modeled as failed when it is in state 2, and it is repaired when it comes to state 1. This happens at the rate 2μ. Now, the mean time to failure of the subsystem, $MTTF_{pair}$, is given by using equation 7.1.

We can extend the above two component subsystems with one redundant component to a k component subsystem with k-1 redundant components. This subsystem fails only when none of its components is active. It can be modeled by a k module, k repairmen model (or an M/M/k/k/k queue). Availability of this system, $A_{k\text{-}redundant}$, can be given by extending equation 6 as follows.

$$A_{k-redundant} = \frac{(\lambda + \mu)^k - \lambda^k}{(\lambda + \mu)^k} \tag{7.7}$$

Practically, this availability quickly approaches 1 for the values of the mean time to failure given in Table 7.3. We can show that the mean time to repair for this subsystem is inversely proportional to k. Corresponding value of the mean time to failure of the subsystem will be very large as compared with the mean time to failure of the individual component.

7.5.1.2.1. Availability of Hardware Components with Redundancy

Now let us look at the effect of some of the hardware fault tolerance techniques discussed in section 7.3.2. Table 7.4 gives the parameters of interest.

$MTTF_{disk}$	mean time to failure of a single disk
$MTTF_{proc}$	mean time to failure of a single processor
$MTTF_{mem}$	mean time to failure of a single memory
$MTTF_{int}$	Mean time to failure of a single interconnect
$MTTF_{ps}$	Mean time to failure of a single power supply

Table 7.4: Component Fault Tolerance Parameters

Effect of Disk Mirroring: Disk mirroring is used to tolerate hard failures of disk units. By disk mirroring, the availability of the disks can be greatly improved.

However, this fault tolerance is achieved at the cost 100% overhead in number of components.

Example: Consider a subsystem consisting of mirrored disks with $MTTF_{disk}$ to be 10K hours and the mean time to repair, $MTTR_{disk}$ to be 24 hours. By using equation (6), availability of a disk unit of mirrored disks, A_{du} evaluates to 0.999942675 and the mean time to failure, $MTTF_{du}$ to 239 years.

Example: A duplication scheme can also be used for memory. For $MTTF_{mem}$ of 100K hours, the mean time to failure of a memory unit consisting of mirrored memory will be 23,793 years! Similar cases arise for dual interconnect and dual power supply. The effect of using dual interconnect and dual power supply can be calculated similarly.

Effect of Processor Pairing: The use of redundancy in processor pairing as discussed in section 7.3.2 is quite different from that in disk mirroring. This is because disk mirroring is used to tolerate hard faults, while processor pairing is used to tolerate soft faults. By processor pairing, all the soft faults are detected. Since many of the errors are transient, many of the faults do not recur. For recurring soft faults, the transaction is aborted or the system code is reinitialized. If the soft error is persistent after several retries, it is manifested as a hard failure. The mean time to failure corresponding to the soft faults is estimated to be 10K hours or about ten times as frequent as the hard failures. The processor pairing tolerates these soft faults. For any hard failure, an interrupt is generated and the processor pair detaches itself from the system. Thus, unlike disk mirroring, the processor pair does not continue to work if one of the two processors in a pair fails.

Example: For a processor pair to be active, both the processors should be active. Thus if mean time to failure of each processor, $MTTF_{processor}$, is 100K hours, then by using equation 7.5, mean time to failure of a processor unit, $MTTF_{pu}$, consisting of processor pair will be 50K hours.

Effect of Processor Redundancy: Processor redundancy is used to tolerate hard failures that disable processor units (which may be a single processor or a processor pair). This is done by using multiple processor units in a cluster. Since at least one processor unit should be active for the cluster to be active, processor redundancy decreases the probability of a cluster failure due to processor failures. Processor redundancy also contributes to an increase in the processing power.

Example: If a cluster contains two processor units, such that each unit is a processor processor pair with the mean time to failure of each unit, $MTTF_{pu}$ to be 50K hours, then the mean time to failure of the cluster *due to processor failures*, $MTTF_{procs}$ will be 5950 years. If a cluster contains three processor units, $MTTF_{procs}$ will be 8,269,650 years!

7.5.2. Parameters

Many parameters affect the availability of a system. Table 7.4 defines component parameters. Table 7.5 defines system parameters. The last three parameters in Table 7.5 are called output parameters; the rest are called input parameters. Range or

multiple values of input component and system parameters are used to study the effect of parameter values on system availability. Table 7.6 gives range values and default values used in our quantitative evaluations. If the value of a parameter used during a calculation is not explicitly mentioned, then the default values should be assumed.

NC	Number of clusters in the system
ND	Number of disk units per cluster
NP	Number of processor units per cluster
DBS	Database size
DF	Distribution factor
C	Interconnect capacity
PS	Page size
I	I/O access time
$MTTF_{cluster}$	Cluster MTTF
$MTTR_{cluster}$	MTTR for a cluster failure
Av	System availability

Table 7.5: Important System Parameters

The distribution factor (DF) is given by the average number of clusters on which the second copy of data stored on one cluster is stored. For example, the DF of the data placement scheme shown in Figure 7.2 is 2, and the DF of the data placement scheme shown in Figure 7.3 is 1. I/O access time is the time to transfer one page from a disk to memory.

Parameter	Values/Range	Default
$MTTF_{disk}$	10K or 30K (hours)	30K (hours)
$MTTF_{proc}$	100K (hours)	-
$MTTF_{mem}$	100K (hours)	-
$MTTF_{int}$	100K (hours)	-
$MTTF_{ps}$	100K (hours)	-
NC	1024 to 2	-
ND	1 to 512	-
NP	1, 2 or ND/2	ND/2
DBS	10**11 (100 Giga) or 10**12 (1 Tera) byte	10**12 (byte)
DF	1, NC/2 or NC-1	NC/2
C	100M, 500M, or 1000M (bits)	500M (bits)
PS	4K, 16K, or 64K (bytes)	16K (bytes)
I	20 ms	-

Table 7.6: Input Parameter Values

Depending on the component fault tolerance techniques used in a system, we get different system configurations. We identify three system configurations of interest to

our study. These configurations are given in Table 7.7.

Configuration 1	Single disk, processor pair, single memory, single interconnect, single power supply
Configuration 2	Mirrored disk, processor pair, single memory, single interconnect, single power supply
Configuration 3	Mirrored disk, processor pair, dual memory, dual interconnect, dual power supply

Table 7.7: System Configuration

To simplify the presentation without reducing the scope of evaluation, we will assume that all the clusters are identical and all the components of the same type have the same characteristics. We also assume that the total number of disk units in the system will be 1024 (i.e., $NC * ND = 1024$). If each disk unit uses mirroring, there will be twice as many disks in the system. We will assume the data storage scheme described in section 7.3.3.3. Because this scheme uses data duplication, the total amount of storage capacity required in the system will be twice that of DBS. Thus if the DBS is 100 gigabytes, then each disk unit should have a storage capacity of (100 * 2 / 1024) or nearly 200 megabytes. If the DBS is 1 terabyte, then each disk unit should have a storage capacity of nearly 2 gigabytes.

In this study we focus only on the fault tolerance. Thus we do not consideration in detail the query processing and optimization issues. However, it should be noted that these issues are very dependent on the data storage scheme used. Fault tolerance is also very dependent on the data storage scheme used. But a data storage scheme that is good for fault tolerance may not be very good for query processing. This presents a very important trade-off that we do not address in this study.

7.5.3. Mean Time to Failure of a Cluster

A cluster is active if all of the following hold true:

1. All of the ND disk units are active,

2. At least one of the NP processor units is active,

3. Shared memory unit is active,

4. Interconnect and connection to it is active, and

5. Power supply unit is active.

Thus, by using equation 7.1, $MTTF_{cluster}$ is given by:

$$\frac{1}{MTTF_{cluster}} = \frac{1}{MTTF_{disks}} + \frac{1}{MTTF_{procs}} +$$

$$\frac{1}{MTTF_{mu}} + \frac{1}{MTTF_{int}} + \frac{1}{MTTF_{psu}} \qquad (7.8)$$

where $MTTF_{disks}$ is the MTTF of the system of all the disk units in the cluster, $MTTF_{procs}$ is the MTTF of the processor units in the cluster, $MTTF_{mu}$ is the MTTF of the shared memory unit in the cluster, $MTTF_{int}$ is the MTTF of interconnect and connection to it, and $MTTF_{psu}$ is the MTTF of power supply unit in the cluster. Each of these items are discussed below.

- All of the disk units of a cluster must be active in an active cluster. This is because of the way we distribute the data, namely, the data allocated to a cluster is vertically fragmented and different fragments are stored on different disks. Hence all the fragments are required to construct a copy of a relation. $MTTF_{disks}$ can be given by equation 7.5 because it is a system consisting of k identical disk units working in parallel and each of the units should be available for all the system to be available. Thus:

$$MTTF_{disks} = \frac{MTTF_{du}}{k}$$

where $MTTF_{du}$ is the MTTF of a disk unit. Each disk unit may be comprised of a single disk or mirrored disks. Section 7.5.1 discusses how to calculate MTTF for mirrored disks.

- At least one of the processors units must be active in an active cluster. When a processor fails, transaction fault tolerance aborts the transactions being executed by the failed processor. As long as at least one of the processor units is active, the processing will continue in the cluster, but at reduced speed. Such a property of the system is called graceful degradation. A processor unit is either a single processor or a processor pair. Subsection 5.1 discusses how to calculate MTTF when a cluster contains several processor units.

- The shared memory must be active in an active cluster. The shared memory could be either nonredundant or redundant. If it uses memory mirroring, $MTTF_{mem}$ can be calculated in a way similar to the mirrored disks.

- An interconnect and the cluster's communication to it must be active for an active cluster. The system may have single interconnect or dual interconnects (i.e., two interconnects with independent communication controllers at each cluster). $MTTF_{int}$ of a dual interconnect can be calculated as in the case of mirrored memory.

- The power supply must be active for the cluster to be active. A cluster may have a single or a dual power supply. $MTTF_{psu}$ can be calculated as in the case of dual interconnect and mirrored memory.

7.5.3.1. MTTF Results

There are many parameters that influence $MTTF_{cluster}$. Among the important ones are the size of the cluster (primarily identified by ND), fault tolerance techniques used for each of its components (i.e., configuration), the reliability (or mean time to failure) of each of its components, and the data storage scheme. As noted in before, we quantitatively evaluate only one data storage scheme, which we discussed in section 7.3.3.3.

The larger the size of a cluster, the more parallelism and storage capacity within a cluster. However, there are more components in a cluster that can fail. Thus the $MTTF_{cluster}$ will decrease. Two important parameters that decide cluster size are ND and NP. In terms of fault tolerance and our study, ND plays a significantly more dominant role because of the following reasons:

- $MTTF_{disk}$ is smaller than $MTTF_{processor}$ because a disk has mechanical components and hence is inherently less reliable.

- Usually ND is larger than NP since the memory contention limits the number of processors that can be used in parallel.

- ND also decides the size of the database.

Because of the importance of ND and the sensitivity of all the three output parameters with respect to it, we plot it on the x-axis for all the graphs. The output parameters are plotted on the y-axis.

The fault tolerance techniques require redundancy and hence more components. The system configurations reflect the hardware fault tolerance techniques used in the system. Figure 7.7 compares the $MTTF_{cluster}$ of cluster using different configurations. Configuration 3, which uses hardware fault tolerance techniques for all of its components, performs significantly better.

Because the MTTF of a disk is significantly smaller than the MTTF of other components, $MTTF_{disk}$ has a very significant effect on $MTTF_{cluster}$. This effect is shown in Figure 7.8. Note that the scale on the y-axis is logarithmic (i.e., it doubles every unit).

Figure 7.9 compares $MTTF_{cluster}$ of a cluster that has one processor with a cluster that has more than one processor. The difference between the curves for $NP = 1$ and $NP \geq 2$ is significant for smaller values for ND. At larger values for ND, $MTTF_{disks}$ becomes a bottleneck for both cases; so the difference is not significant. Also, having more than two processors does not increase the $MTTF_{cluster}$ noticeably because the value of $1 / MTTF_{procs}$ does not contribute significantly in equation 7.8.

7.5.4. Response Time of a Cluster Recovery

A cluster recovery takes place after a cluster fails. In section 7.3.3.4, we discussed a procedure to perform such a recovery. Estimating $MTTR_{cluster}$, response time for cluster recovery is quite difficult. As explained below, we will calculate an optimistic estimate.

The step of the recovery procedure that may contribute the most to the recovery time is step 5, "send the data to the destination cluster." This step can be divided into three substeps.

Step 5a. Read the data from the source clusters. The DF is the same as the number of source clusters; so the recovery data will be read from the disks of each of the source clusters. Because we assume that vertical fragments of the parts of relations are stored on a cluster, it is possible to retrieve data from all the disks in a cluster in parallel. The following equations give the time to read the recovery data.

$$\text{total data lost due to cluster failure, } DBC = 2 \times \frac{DBS}{NC}$$

$$\text{total recovery data on a source cluster, } DBR = \frac{DBC}{DF}$$

$$\text{data on a disk in source cluster, } DBD = \frac{DBR}{ND}$$

$$\text{number of pages to be read from a disk, } PG = \frac{DBD}{PS}$$

$$\text{time to read PG pages or the total read time, } RT = PG \times I$$

Step 5b. Transmit data from source clusters to destination cluster. We assume that the data received in a one page fetch cycle is packeted together and transmitted to a destination cluster. Transmission time includes the time required for physical channel transmission and communication software overhead. Optimistically, we assume a communication software overhead factor (CSF) of 2.0. The time to transmit recovery data can be calculated as follows.

$$\text{packet size, } PKS = ND \times PS$$

$$\text{time to transmit a packet, } TP = \frac{PKS}{C} \times CSF$$

$$\text{total transmission time, } TT = TP \times PG$$

Step 5c. Write data on the destination cluster. The time to perform this step can be calculated as in step 5a.

We assume that as far as possible, the system will read, transmit and write data in parallel. Thus, if $I < TP$ (i.e., the time to read a page is less than the time to transmit a packet), the total time for step 5 is $(I + TT + I)$, where the first I is due to the time to read a page and the second I is due to the time to write the data received in a packet in parallel on all the disks in a destination cluster. On the other hand, if $I > TP$, then the total time for step 5 is $(TR + TP + I)$.

Table 7.8 summarizes the time taken by various steps of the recovery procedure. The time taken for other steps of the recovery procedure is our best guesstimate.

	Step of Recovery Procedure	Time Taken
1.	Detect failure	2 sec
2,3,4.	Prepare to send data	10 sec
5.	Send data	max(2I + TT, TR + TP + I)
6,7.	Update dictionary and restart	2 sec

Table 7.8: Factors Contributing to MTTR

7.5.4.1. MTTR Results

As in case of $MTTF_{cluster}$, $MTTR_{cluster}$ is influenced by many parameters. Among the important parameters are size of the database, distribution factor, interconnect capacity and page size.

The larger the size of the database, the more data will become unavailable when a cluster fails. Thus more data will need to be transferred during the recovery procedure especially if the system has fewer large clusters. Figure 7.10 shows that $MTTR_{cluster}$ is significantly larger for database size of 1 terabyte as compared to the database size of 100 gigabytes. Most of the $MTTR_{cluster}$ is contributed by the transmission time (TT).

A higher DF means more clusters have the recovery data and hence less recovery data per cluster and disk in a source cluster. Thus it is possible to have more parallelism for systems with a higher DF. $DF = 1$ means that only one cluster has all the second copy of data required for recovery, $DF = NC / 2$ means on average half of the cluster in the system have part of the recovery data, and $DF = NC-1$ means all the active clusters have some part of the recovery data. Figure 7.11 shows that effect of the DF is very significant. Since the DF depends on the data storage scheme, the effect of data storage scheme on $MTTR_{cluster}$ is very significant.

We found that TP > I in most cases. Thus the transmission time is more dominant than read time, especially for large clusters (i.e., higher ND). Because of this, effect of interconnect capacity, C, on $MTTR_{cluster}$ is significant, particularly for clusters with $ND \geq 16$. See Figure 7.12.

The larger the page size, the more recovery data is read in a page fetch cycle. This results in smaller I/O read and write time and comparatively larger transmission time per packet. For smaller clusters, the read time is dominant and since large page size results in fewer page fetches, a system with larger PS has a smaller $MTTR_{cluster}$. See Figure 7.13. For $ND \geq 64$, transmission time dominates; so the PS does not affect $MTTR_{cluster}$.

7.5.5. System Availability

The mean time to a cluster failure in a system is $NC * MTTF_{cluster}$. The mean time to repair from such a failure is $MTTR_{cluster}$. Thus the system availability can be calculated as follows using Equation (1).

$$AV = \frac{(NC \times MTTF_{cluster})}{(NC \times MTTF_{cluster} + MTTR_{cluster})} \tag{10}$$

7.5.5.1. Availability Results

To study the effect of various parameters on a system, we evaluate same-size systems. We do this by fixing the total number of disk units in a system to 1024 (i.e., NC $* ND = 1024$). Thus if the system has large clusters (i.e., ND is large), then the system has fewer such clusters. However, we will need fewer large clusters in a system for a given database size. Depending on the capacity of each of the disk, the total database size can be varied from 100 gigabytes to 1 terabyte. This size, in our opinion, defines a very large database system.

System availability is affected by all the parameters that affect $MTTF_{cluster}$ and $MTTR_{cluster}$. The parameters that have more significant effect are system configuration, $MTTF_{disk}$, NP, DBS, DF, C and PS.

Table 7.9 relates Av to system down time. The desired level of availability depends on the type of system application. We feel that modern systems will be expected to give availability of 0.999 or more. It is comforting to note that the fault tolerance techniques we already know may be able to give such availability even for very large systems. It may be noted that all the Av curves that follow are drawn to logarithmic scale. Thus towards the higher end of the scale, even small vertical separation of curves may mean very significant differences in Av.

Av	Unavailability of the System
0.99	Unavailable for one hour in 4 days
0.999	Unavailable for one hour in 41 days (more than a month)
0.9999	Unavailable for one hour in 416 days (more than a year)
0.99999	Unavailable for one hour in 4,166 days (more than 11 years)
0.99999	unavailable for one hour in 41,666 days (more than 114 years)

Table 7.9: Availability vs Unavailability

Figure 7.14 compares the Av of different configurations This shows that the effect of system configuration (i.e., the hardware fault tolerance techniques) is very significant. For example, a 64-cluster system with 8 disk units each will be unavailable for one hour in approximately 79 days if it uses configuration 1, unavailable for 1 hour in approximately 767 days (2.1 years) if configuration 2 is used, and unavailable for 1 hour in 1250K hours (143 years) if configuration 3 is used.

Figure 7.15 compares Av for systems with different $MTTF_{disk}$. Since $MTTF_{disks}$ is always the bottleneck for configurations 1 and 3 and usually the bottleneck for configuration 2, the effect of this parameter is very significant. It should, however, be noted that because both configuration 2 and 3 use disk mirroring, the improved fault tolerance results in a 100% increase in disk costs.

Figure 7.16 shows the effect of NP on Av. This figure shows the curves for configuration 2, in which $MTTF_{procs}$ is the bottleneck for $NP = 1$. Thus the effect of

processor redundancy is significant in this case. However in all other cases, because disk availability is the bottleneck for Av, the effect of processor redundancy is not significant if $NP \geq 2$. By comparing the curves for $NP = 2$ and $NP = ND / 2$, we also note that the effect of more than two processor on fault tolerance is insignificant. Thus, the primary purpose of having more than two processors in a cluster will be efficiency in query processing and not fault tolerance. Also note that processor redundancy costs less than disk redundancy.

Figure 7.17 shows the effect of DBS on Av. The difference between DBS of 100 gigabytes and 1 terabyte is less when the system consists of many small clusters because of the smaller difference between values for $MTTR_{cluster}$. This difference is very significant for the system consisting of fewer large cluster for both $MTTR_{cluster}$ (see Figure 7.10) and Av.

Figure 7.18 shows the effect of DF on Av. A higher DF means more clusters have the recovery data and take part in recover procedure, thus providing more parallelism. This effect is more significant for a system consisting of a large number of small clusters.

Figure 7.19 shows the effect of C on Av. The differences for a system of many small clusters is insignificant because the time to read recovery data is dominant (i.e., the system is node bound). However, the transmission time is dominant (i.e., the system is communication bound) for a system with few large clusters, so the effect of C is more significant.

Figure 7.20 shows the effect of PS on Av. The effect is more significant for a system with many small clusters. In general, Av is higher for higher PS. A higher PS means fewer pages need to be fetched and larger chunks of data need to be transmitted. This results in the system becoming communication bound sooner.

In general, we can observe that systems with many small cluster have better availability and fault tolerance.
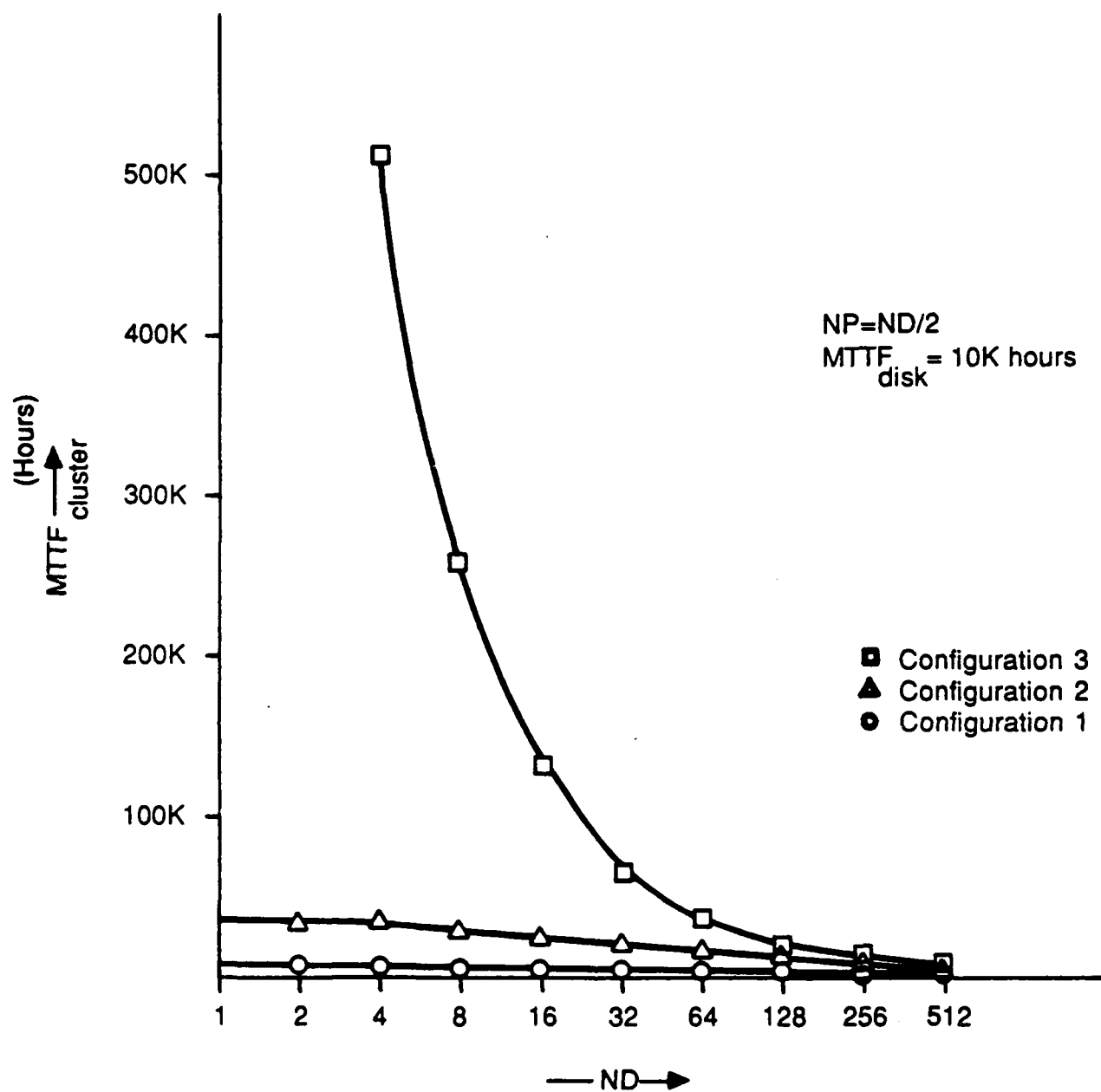
Figure 7.7. Effect of Configuration on $MTTF_{cluster}$
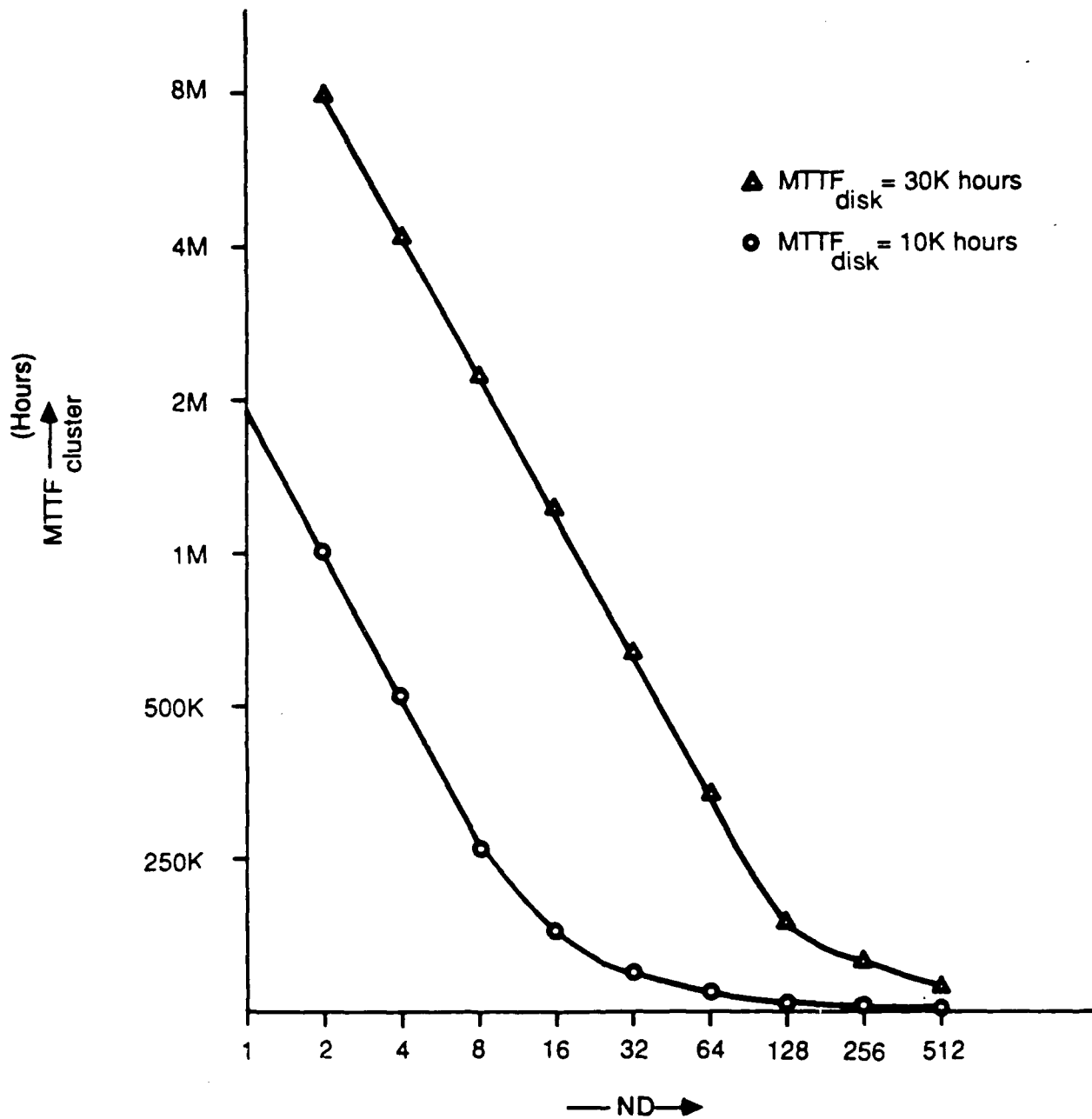
Figure 7.8. Effect of $MTTF_{disk}$ on $MTTF_{cluster}$
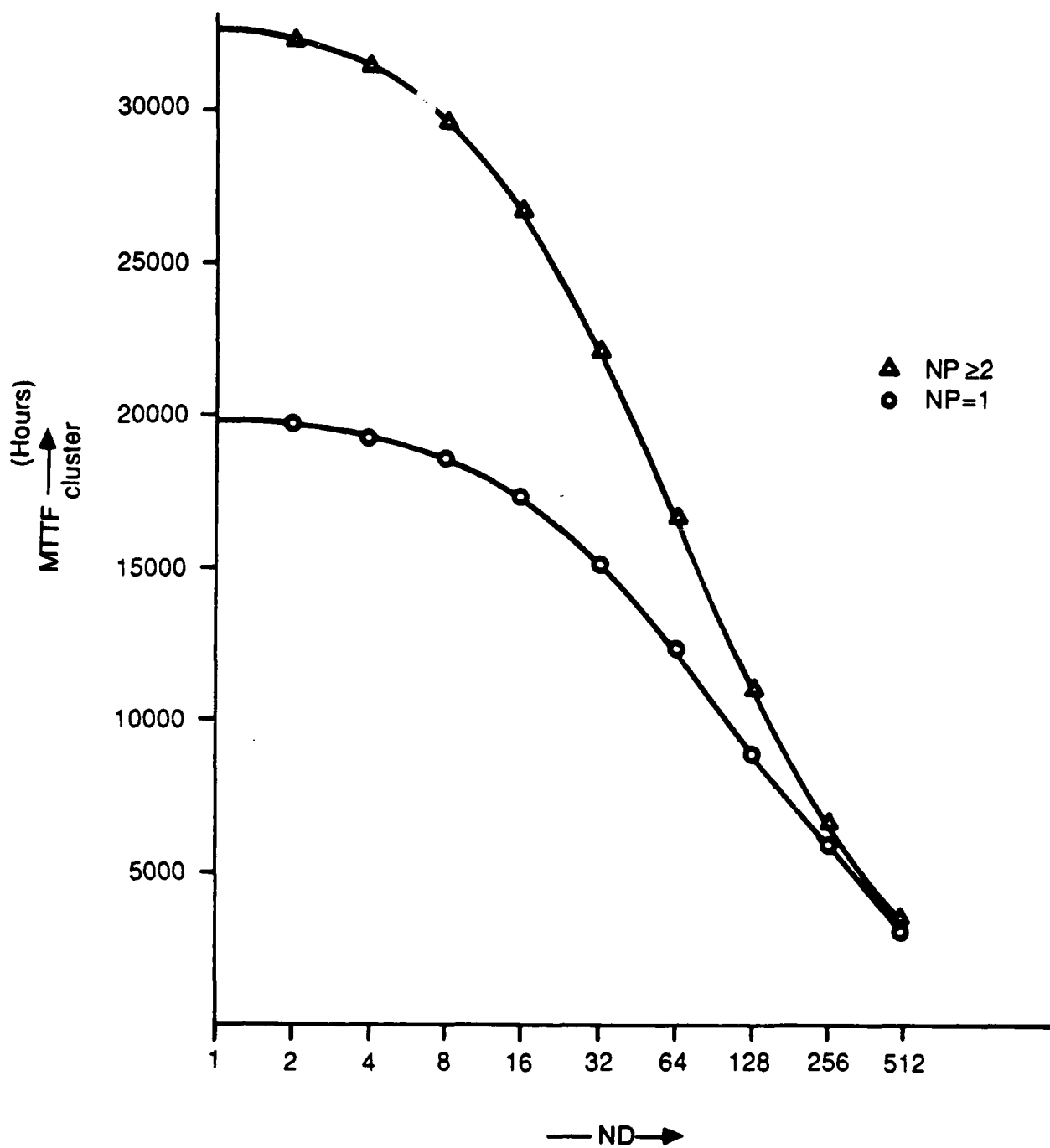
87028-0813M

Figure 7.9. Effect of NP on $MTTF_{cluster}$
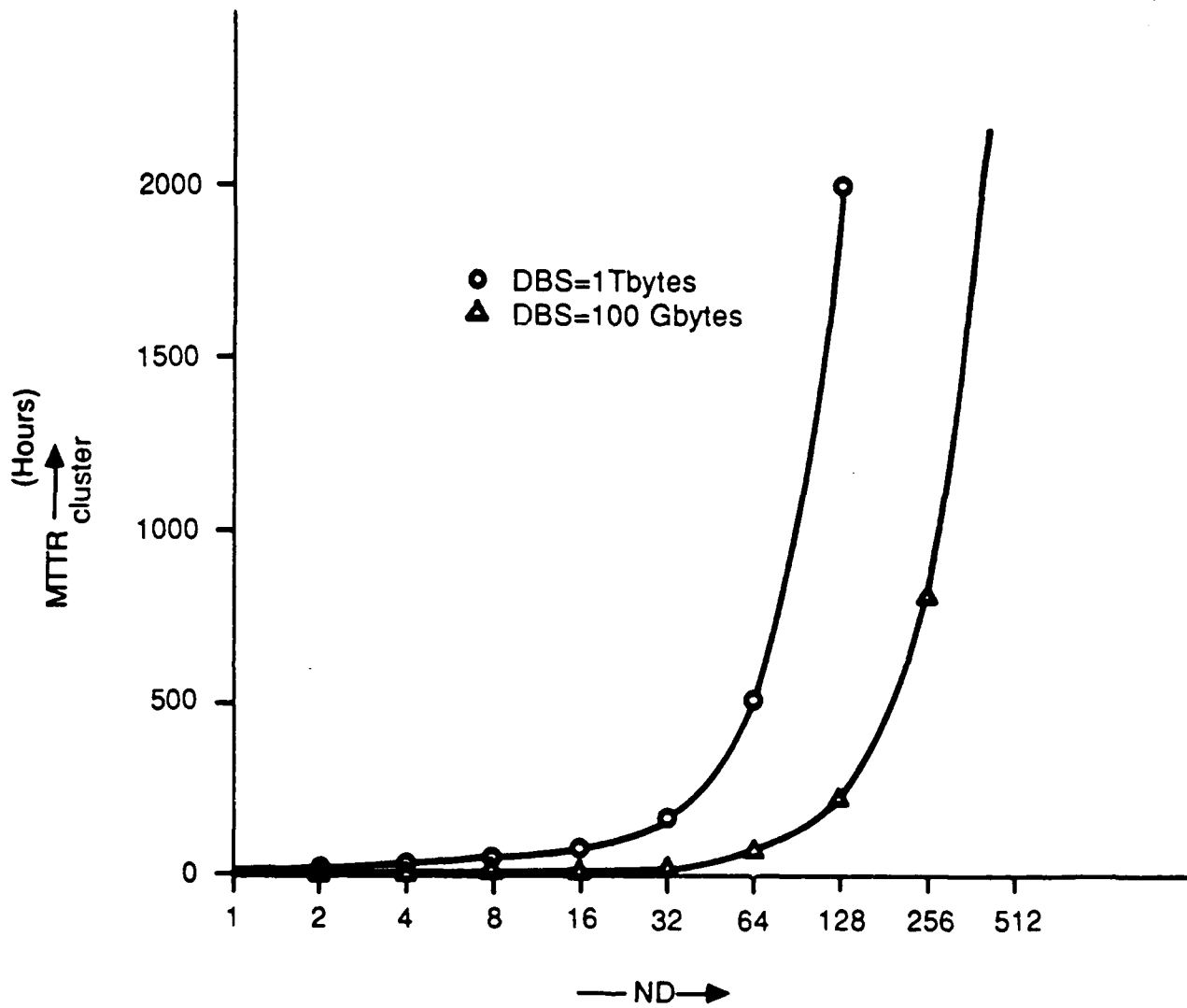
87028-0812M

Figure 7.10. Effect of *DBS* on $MTTR_{cluster}$
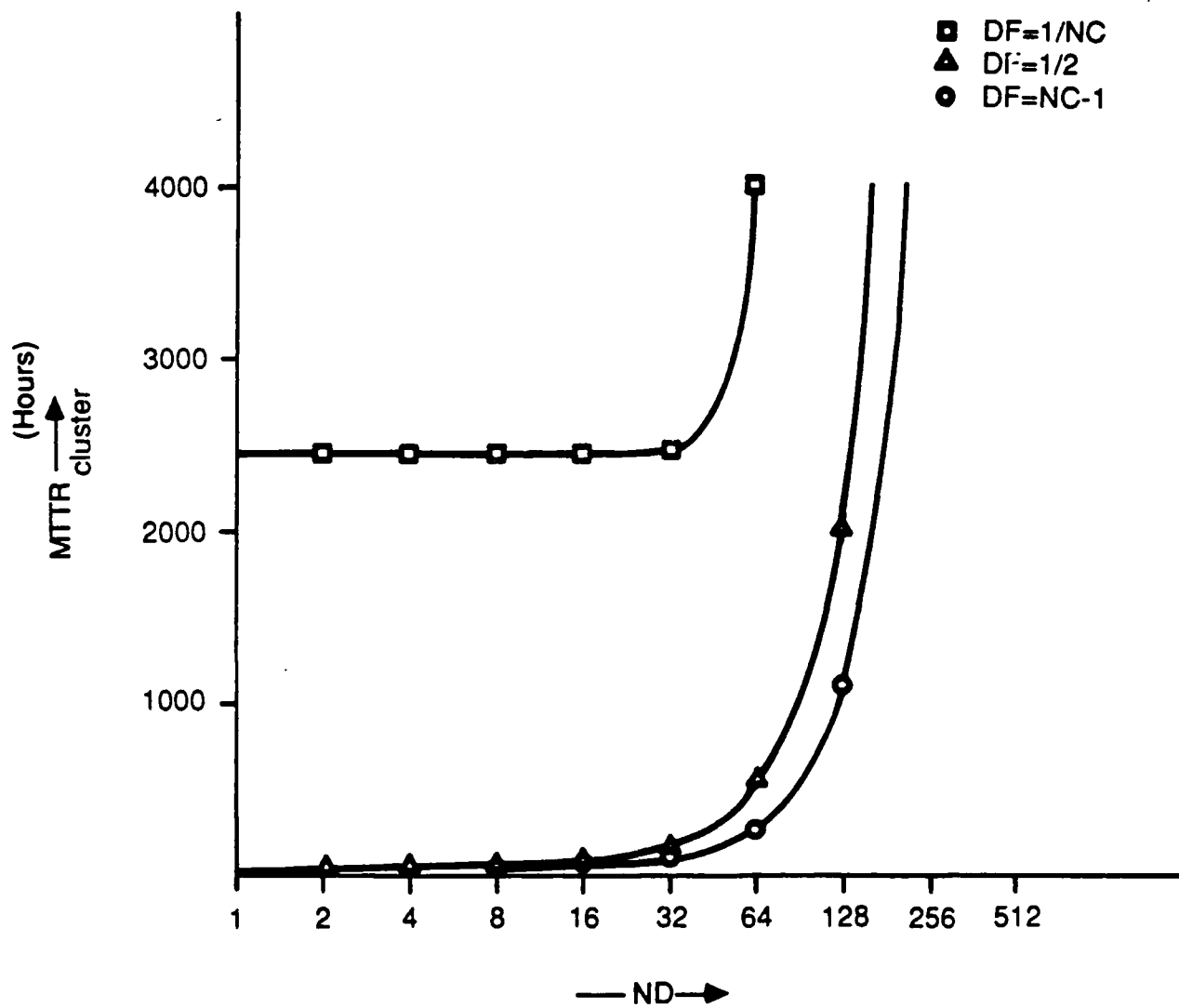
Figure 7.11. Effect of DF on $MTTR_{cluster}$
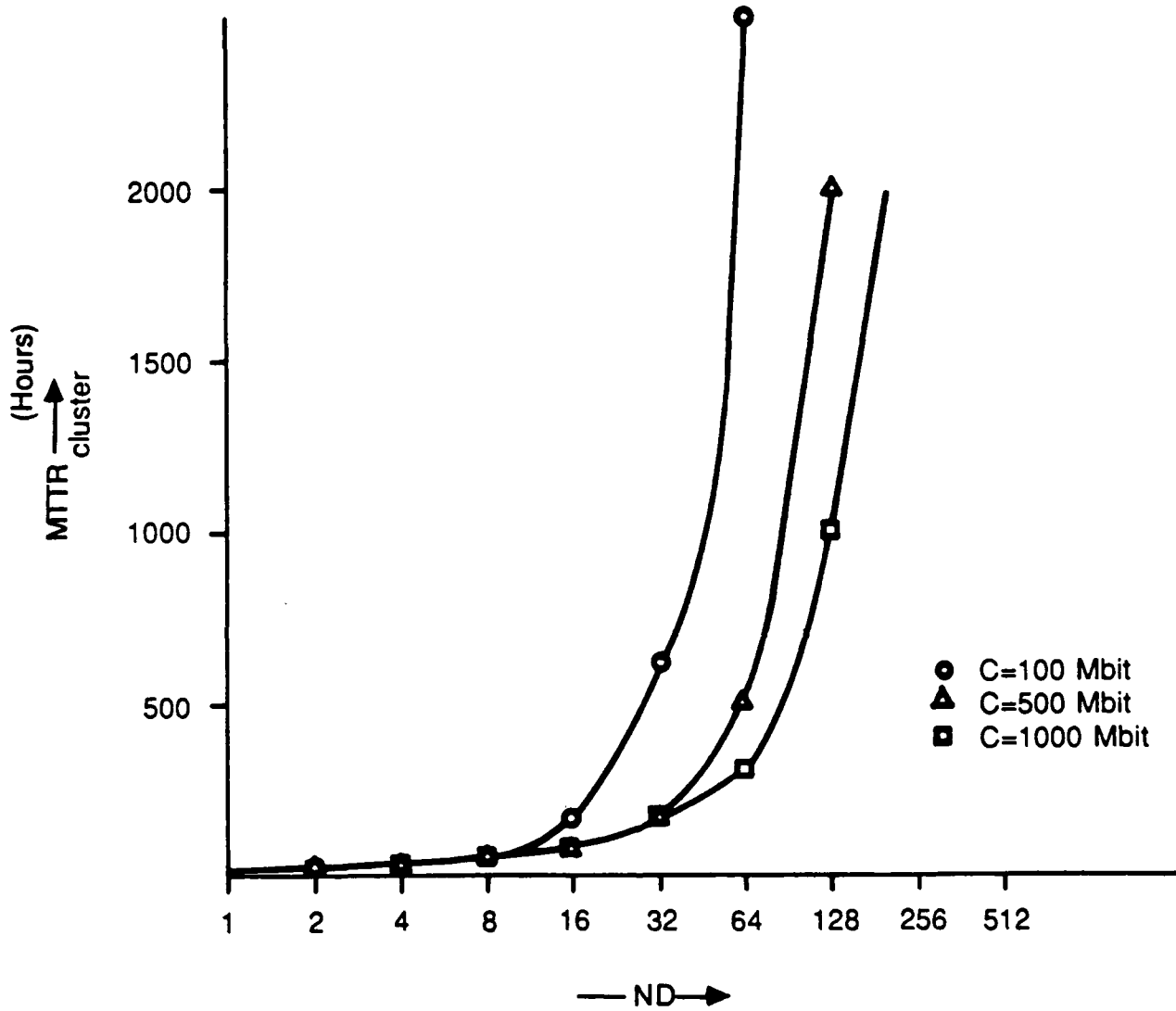
Figure 7.12. Effect of C on $MTTR_{cluster}$

87028-0810M
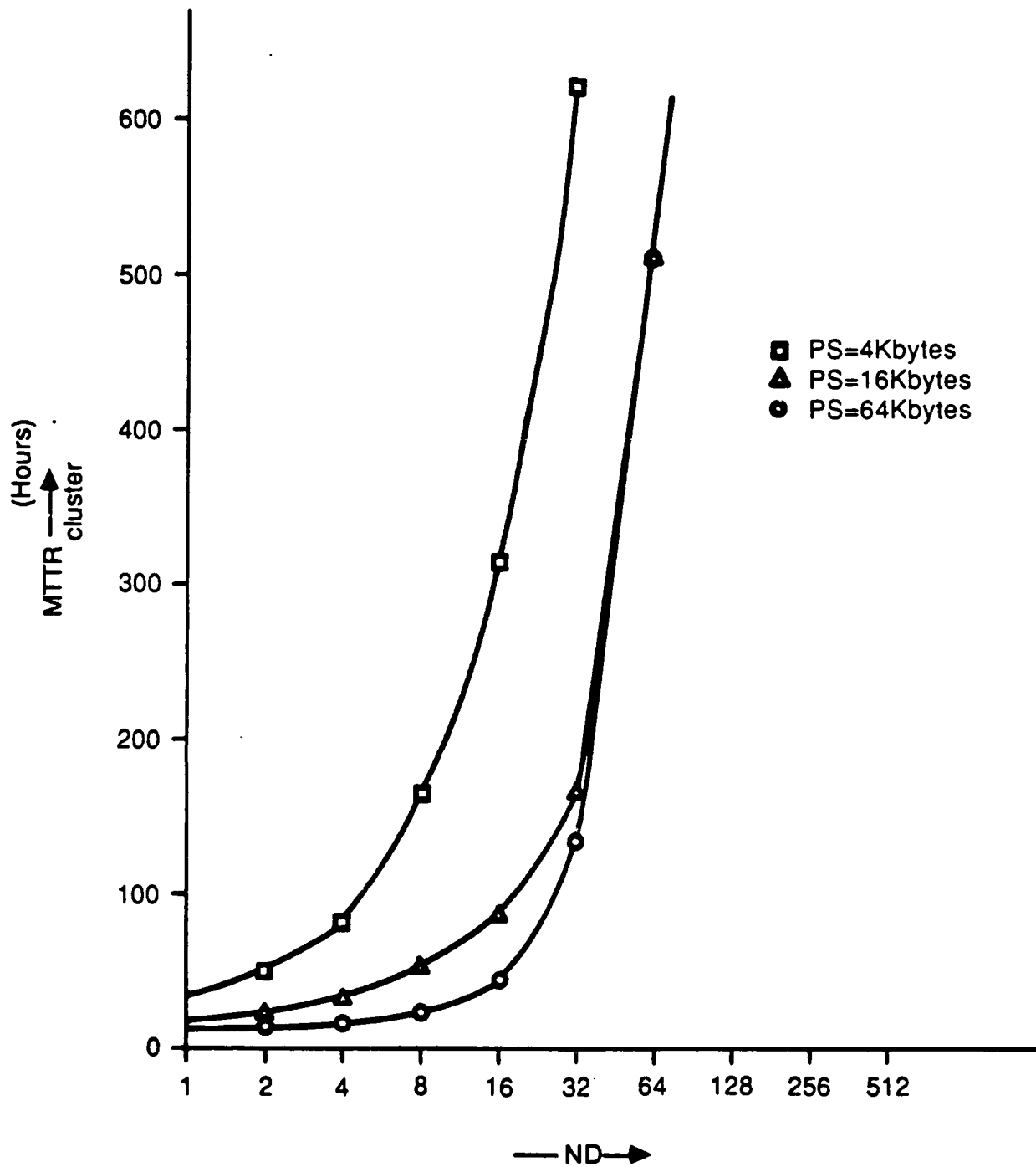
87028-0809M

Figure 7.13. Effect of *PS* on $MTTR_{cluster}$

Figure 7.14. Effect of Configuration on Av
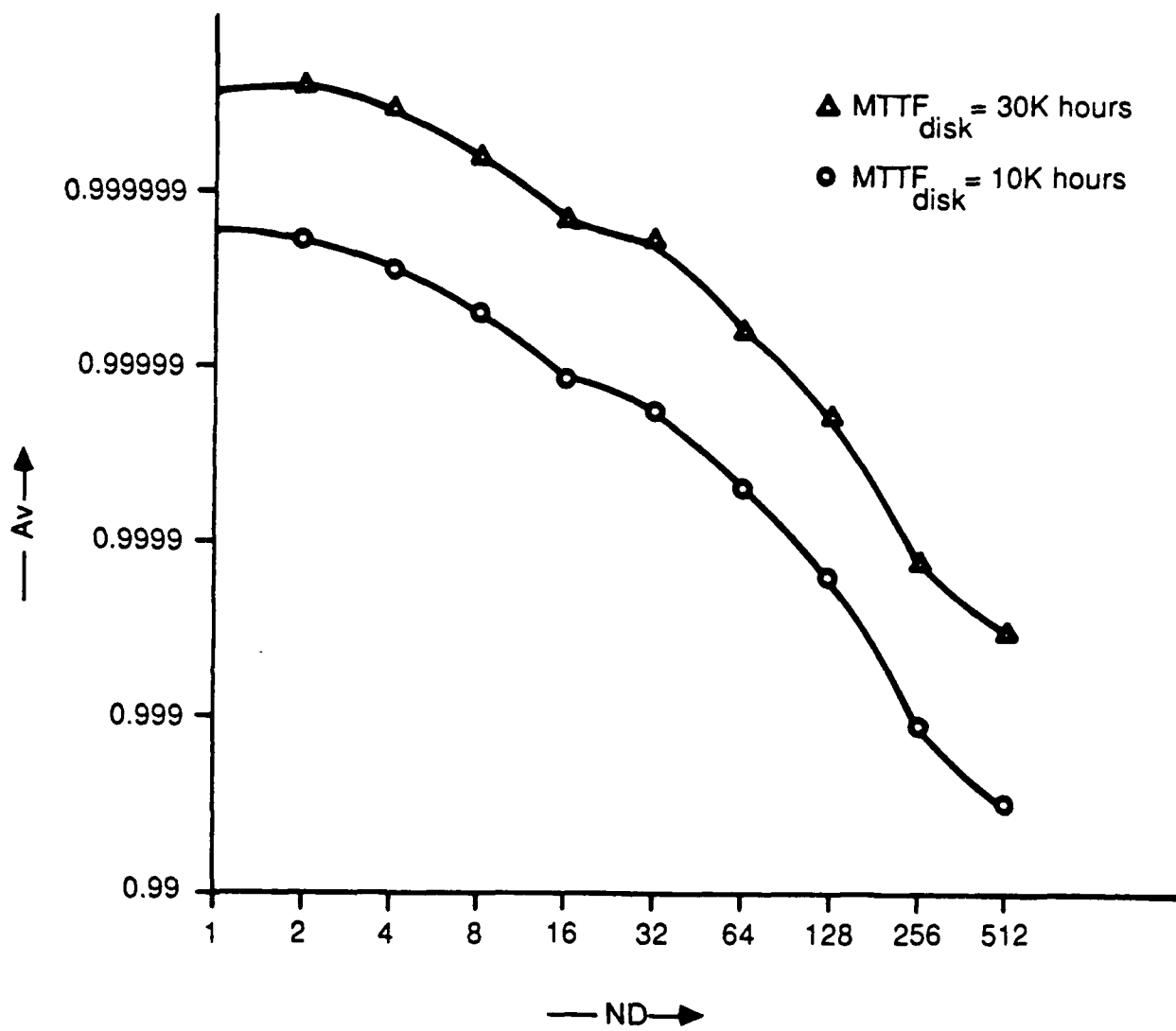
Figure 7.15. Effect of $MTTF_{disk}$ on Av
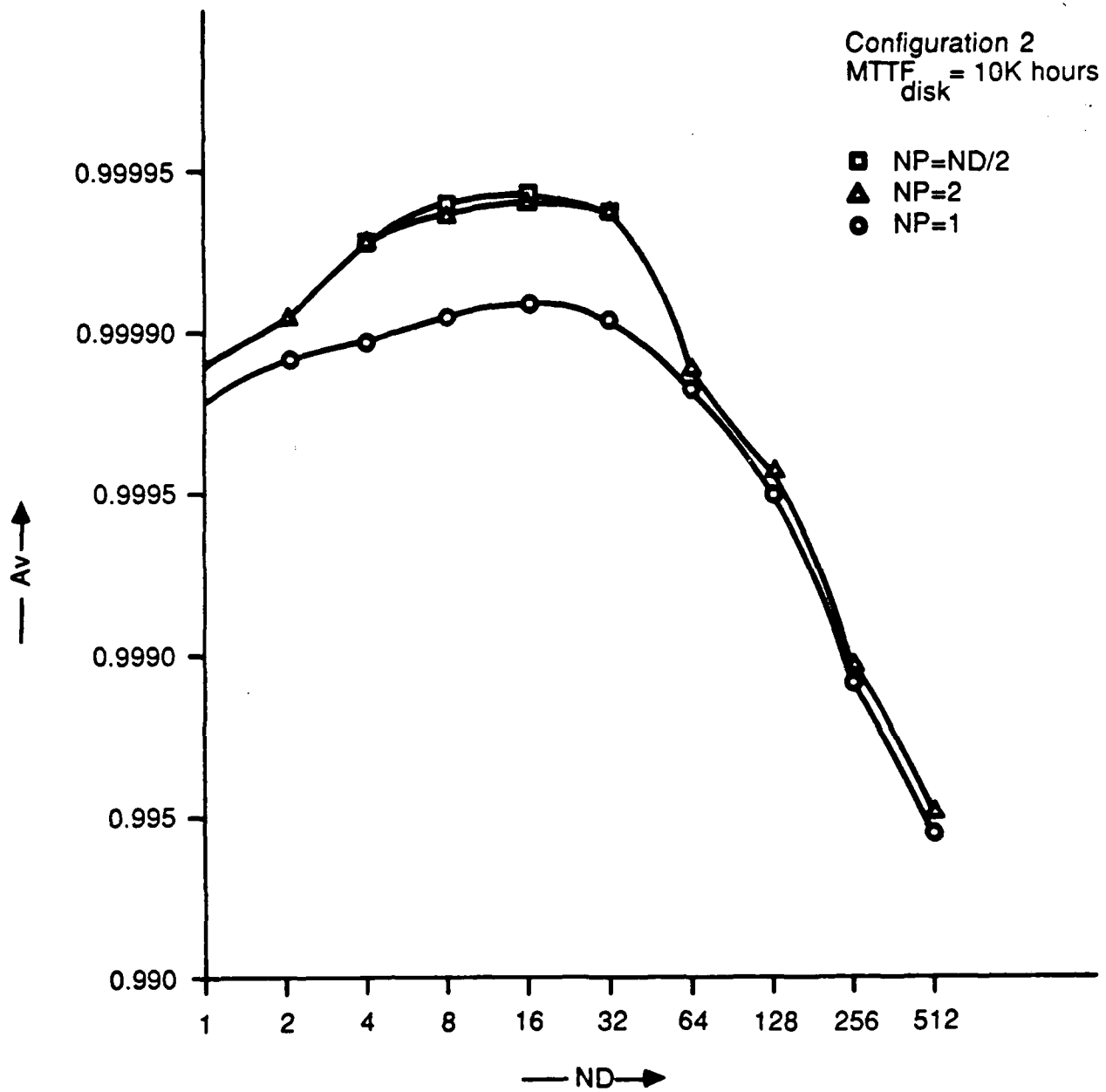
Figure 7.16. Effect of *NP* on *Av*

Figure 7.17. Effect of *DBS* on *Av*
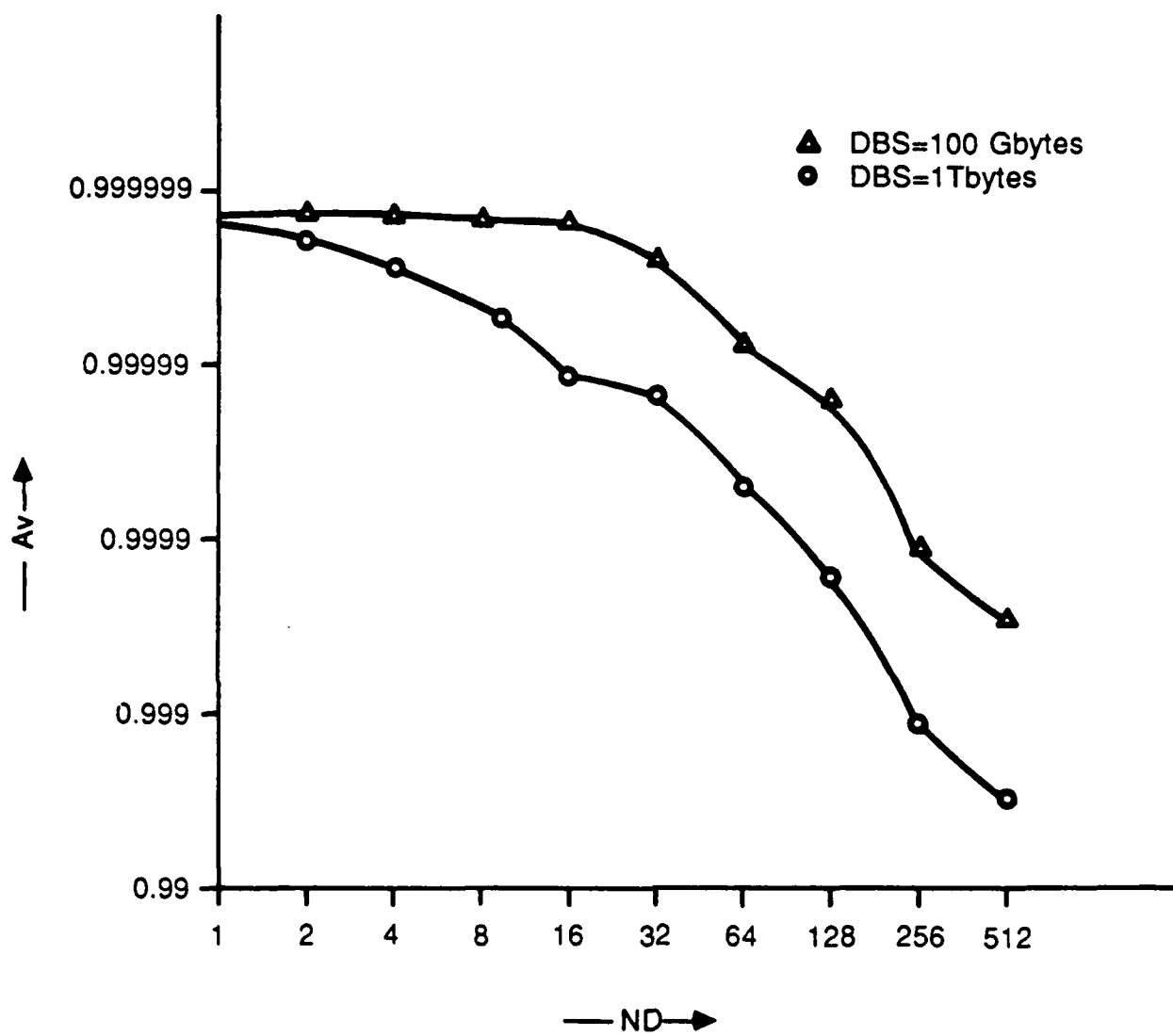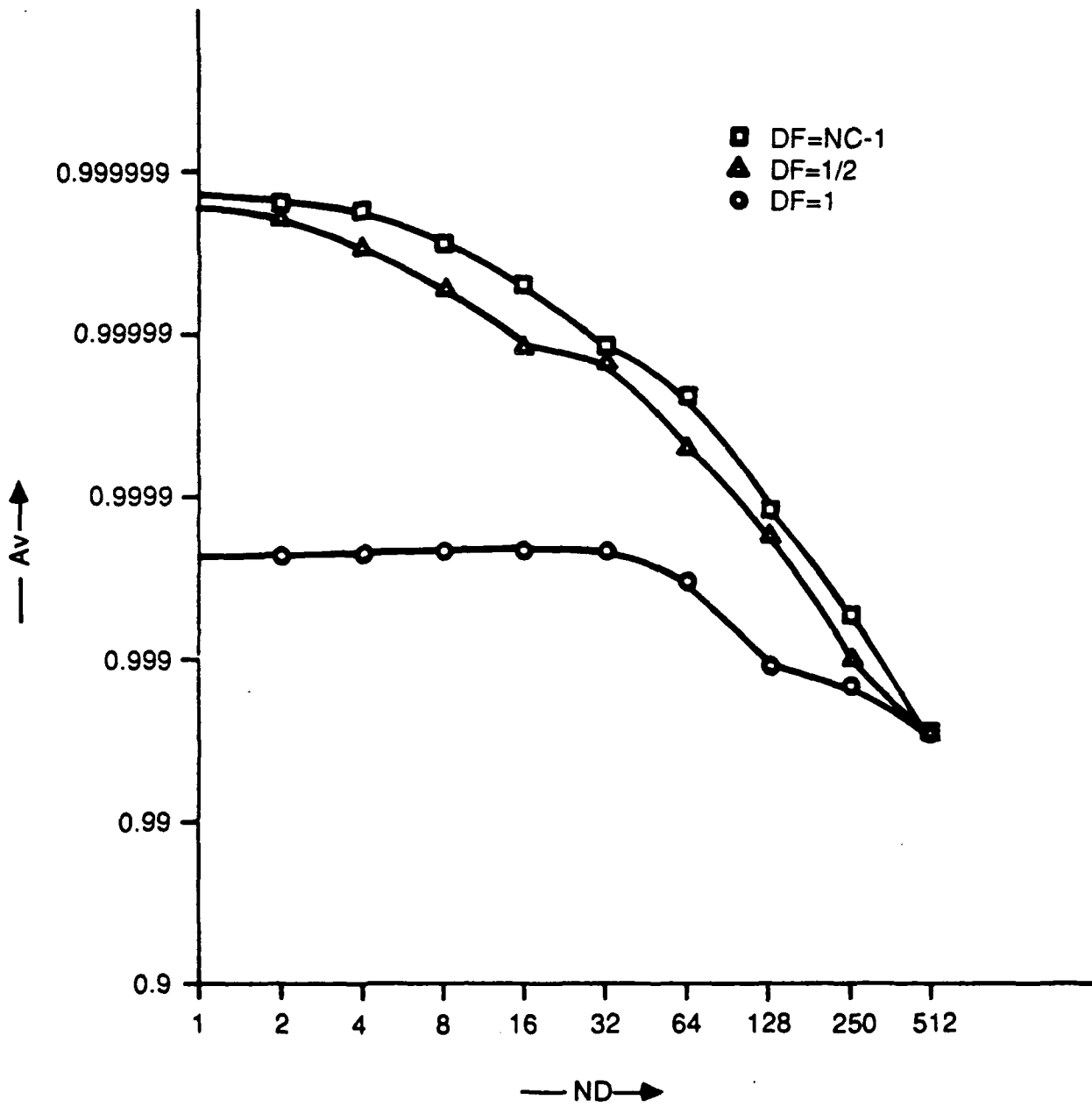
87028-0806M

Figure 7.18. Effect of *DF* on *Av*
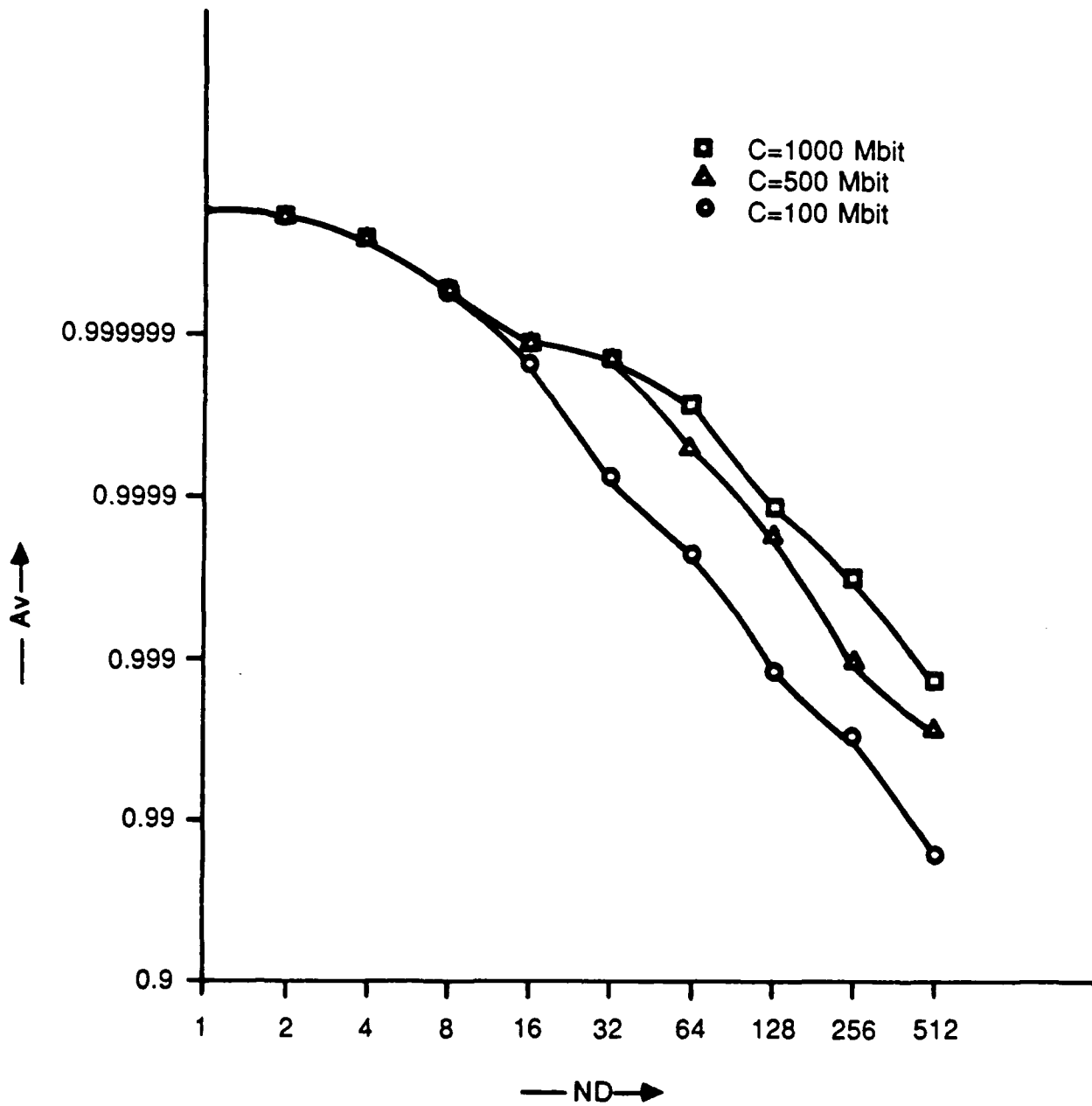
Figure 7.19. Effect of C on Av

PS = 64Kbytes
PS = 16Kbytes
PS= 4Kbytes
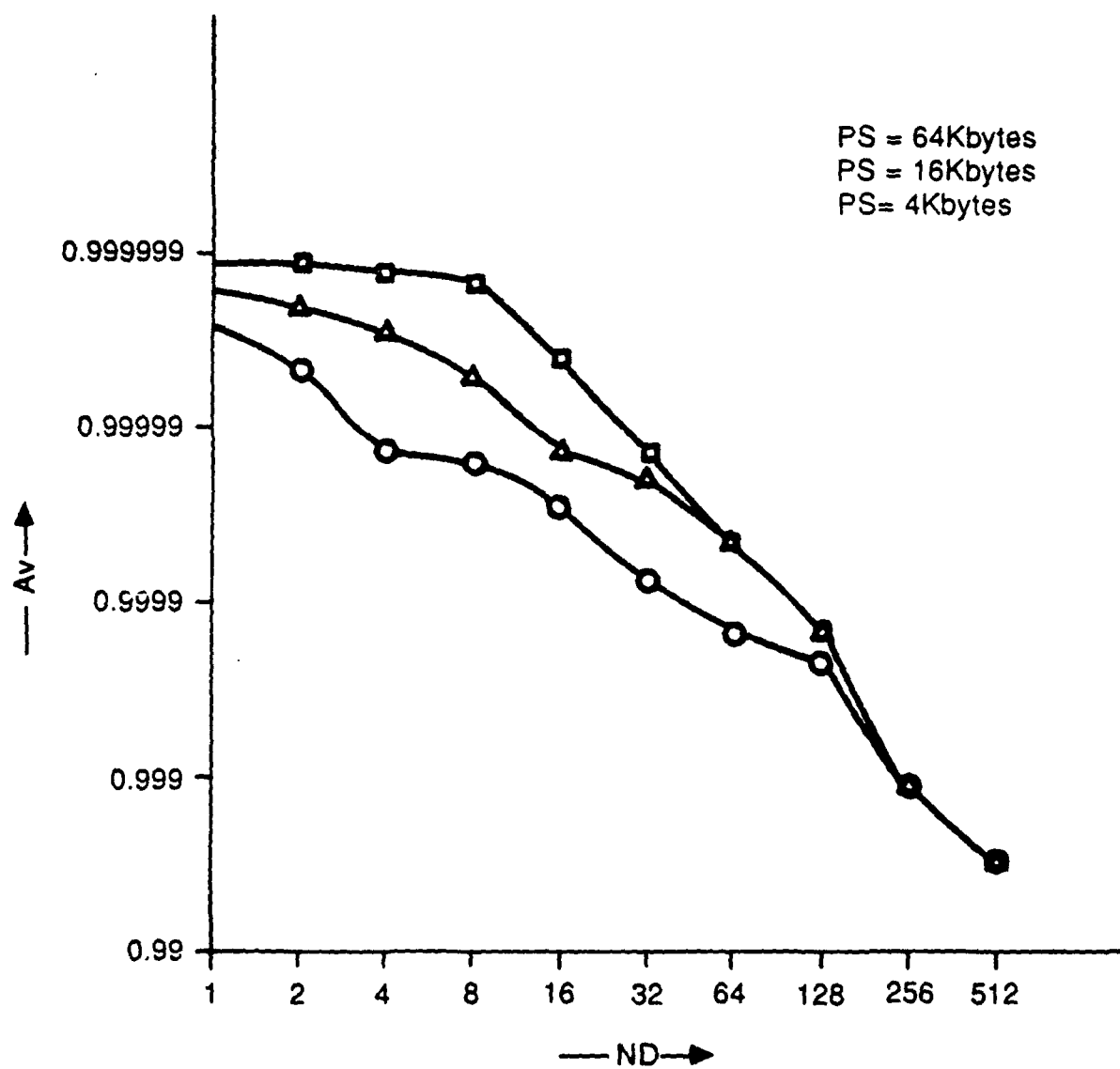
87028-0816M

Figure 7.20. Effect of *PS* on *Av*

7.6. Conclusions

In this chapter, we identified and described basic hardware and software fault tolerance techniques that are required to achieve a high degree of fault tolerance in a very large database system. We also develop quantitative methods to evaluate availability. The quantitative evaluation helps us to understand the relative importance of various parameters that affect system fault tolerance.

We use system availability as the main parameter to measure fault tolerance. Our study shows that system availability is very sensitive to the following parameters:

1. System Architecture. By varying the number of clusters, the number of disks per cluster, and the number of processors per cluster, we were able to study different system architectures. For example, one extreme of our parameterized architecture presents a loosely coupled, non-shared memory architecture (i.e., case of $ND = 1$, $NP = 1$). We found that better availability is obtained when the system has many small clusters. In most cases, $ND \leq 8$ when availability peaks.

2. Fault Tolerance Techniques. We studied the effect of various hardware fault tolerance techniques by varying system configurations that differ in the fault tolerance techniques used for different components. We found that the effect of fault tolerance techniques on system availability is very significant. We also found that reliability of a disk ($MTTF_{disk}$) is usually a bottleneck, and performance gain due to disk mirroring is very substantial. Processor redundancy significantly helps if disks are mirrored (or other disk redundancy methods are used). However, $NP = 2$ is sufficient for fault tolerance. A higher NP does not improve availability significantly. Fault tolerance techniques for other components are useful in conjunction with fault tolerance techniques for disks and processors.

3. Database Size. The size of the database has a significant effect on system availability. If the database is larger, more data will be lost when a hard failure occurs; so recovery takes longer. This degrades availability. Also, as the database size increases, the system will require more components of given capacity for storage and efficient processing. This can have a very significant affect on system availability.

4. Component Reliability and Capacity. Since disk reliability is usually the bottleneck, we studied availability for disks with two different MTTF values. Higher $MTTF_{disk}$ improves availability significantly. System availability will also improve significantly if disks with higher capacities are used (provided $MTTF_{disk}$ of a high capacity disk is not much lower than that for a low capacity disk). We also studied the effect of using interconnects with different capacities. A system with a higher capacity interconnect has a significantly better fault tolerance when the system consists of a few large clusters.

5. Data Storage and Access Method. We studied only a limited aspect of this issue. We used one basic data storage method in our study. By varying the distribution factor, we were able to study how a data storage method can affect recovery time and hence system availability. By showing the dependence of the quantitative

evaluation on the data storage method and by using a distribution factor, the study shows that the data storage method greatly affects system availability. We studied the access method only with respect to page size. A larger page size helps significantly in a system with many small clusters.

CHAPTER 8

D/KBMS Architecture Specification Methodology

Phase I of the VLPDF contract involved six investigation studies: (1) alternative D/KBMS application interface languages, (2) parallel architectures for such languages, (3) D/KB query processing, (4) transitive closure algorithms, (5) parallel D/KBMS architectures, and (6) fault tolerance in very large database systems. During Phase II, the results of these studies were used to develop a methodology for specifying high performance, highly available data/knowledge base management systems for very large data/knowledge base environments. This chapter presents this methodology.

The methodology is presented as a set of policies and steps. The policies constitute a method of action selected from among alternatives to guide and determine D/KBMS design decisions. They are actually a philosophical statement of intent to guide the D/KBMS designer in making suitable design decisions, rather than a comprehensive recipe for design.

Two sets of steps are presented, the first representing a recipe a recipe for D/KB query and update processing, and the second, an overall procedure for D/KBMS architecture specification. The steps for D/KBMS architecture specification are not intended to be comprehensive. Where gaps exist, the D/KBMS designer should consult the policies to determine an appropriate course of action.

The policies are grouped under 10 categories:

1. policies regarding overall D/KBMS functionality,

2. knowledge representation policies,

3. rule storage policies,

4. D/KB query processing policies,

5. D/KB update processing policies,

6. D/KBMS functional partitioning policies,

7. LFP evaluation policies,

8. join processing policies,

9. D/KBMS hardware architectural policies, and

10. fault tolerance policies.

These categories represent the critical issues in the design of high performance, highly available D/KBMSs for very large D/KB environments. Several alternatives are available for addressing each of these issues. These alternatives have a wide ranging performance impact. For example, previously published performance results indicate 6 to 8 orders of magnitude difference in performance between certain LFP evaluation strategies. The policies presented in this chapter are intended to guide the D/KBMS

designer in the choice of suitable alternatives for each critical issue listed above.

The chapter is organized as follows. Sections 8.1 through 8.10 present the policies. Section 8.11 describes the steps in D/KB query and update processing. Section 8.12 presents the overall procedure for D/KBMS architecture specification.

8.1. Policies Regarding Overall D/KBMS Functionality

To motivate the overall D/KBMS functions, let us sketch out a typical user session with the D/KBMS. The user first enters a set of rules and facts. These rules and facts are stored in a memory resident private environment called the *Workspace D/KB*. The Workspace D/KB rules may refer to rules and facts stored in a shared disk resident repository called the *Stored D/KB*. The rules in the Stored D/KB may also refer to rules and facts in the Workspace D/KB. After entering a set of rules and facts into the Workspace D/KB, the user issues queries against them. If he is satisfied that the rules and facts in the Workspace D/KB are correct, he updates the Stored D/KB with these rules and facts.

With this background, we list the overall D/KBMS functions. The D/KBMS shall provide five basic functions: (1) provide knowledge representation and modeling capabilities, (2) enter rules and facts into the Workspace D/KB, (3) enter queries, (4) execute queries, and (5) update the Stored D/KB with rules and facts from the Workspace D/KB.

8.2. Knowledge Representation Policies

Knowledge representation is a basic capability expected of a D/KBMS. This section presents policies relating to knowledge representation.

- The basic level of knowledge representation capability provided by a D/KBMS shall be Horn clause logic. The reason for this is that logic offers several advantages:

  a. it provides a uniform formalism for data, rules, views, and integrity constraints;

  b. it is the basis for relational database theory;

  c. it is amenable to parallel processing;

  d. it is an adequate basis for implementing other knowledge representations; and

  e. it has a sound theoretical foundation, which permits the abstract expression of ideas, independent of their implementation.

- The data/knowledge base shall consist of a set of Horn clauses and schemas. See chapter 4 for the concepts and definitions pertaining to such a data/knowledge base.

8.3. Rule Storage Policies

The choice of rule storage structures is a critical one since it affects the time taken to extract the relevant rules from the Stored D/KB during query processing. This section presents a set of policies relating to rule storage structures.

- The basic rule storage structures in the Stored D/KB shall consist of three relations: *isystables, isyscolumns, and irulesource.* These storage structures are basically "source form" storage structures, in that they contain a direct representation of the source form of the rules.

  The first two relations, *isystables* and *isyscolumns*, shall contain the names and column types of the derived predicates, respectively. These tables shall have the following schema:

  isystables(tablename char, tableid integer)

  isyscolumns(tableid integer, colname char, colnumber integer, coltype integer)

  irulesource shall store for each derived predicate p, the rules defining p, and shall have the following schema:

  irulesource(headpredname char, rule char)

- For update intensive applications, the above source form storage structures shall suffice. However, for query intensive applications, there there shall be a "compiled form" storage structure (described below), in addition to the basic rule storage structures. The motivation for this policy is that with compiled form storage structures, queries can be processed faster, but updates take longer.

- The compiled form storage structure for query intensive applications shall be a relation called *ireachablepreds*. This relation shall be the transitive closure of the PCG of the rules stored in *irulesource.* That is, it shall store for each derived predicate p all the predicates reachable from p. It shall have the following schema:

  ireachablepreds(frompredname char, topredname char).

  The motivation for storing the transitive closure of the PCG is that using this storage structure, the time to extract the relevant rules can be made independent of the total number of rules in the Stored D/KB. If the transitive closure is not stored, it would have to be computed during query processing and the time for doing this increases with the number of rules.

8.4. D/KB Query Processing Policies

This section presents policies relating to D/KB query processing. These policies constitute a broad D/KB query processing strategy; the specific policies relating to high performance query execution are presented in sections 8.7 through 8.9.

- D/KB query processing shall consist of two phases: query compilation and query execution. The motivation for splitting up query processing is twofold. First, several optimizations can be performed during the compilation phase and second, frequently occurring queries can be precompiled, speeding up the processing of such queries.

- During compilation, all the rules needed to solve the query shall be brought into the Workspace D/KB. In general, these rules will be present in both the Workspace and Stored D/KBs. Bringing all of them into the Workspace D/KB will typically involve first determining from the existing Workspace D/KB rules all the predicates reachable from the query and then extracting from the Stored D/KB the rules needed to solve these predicates.

- After all the relevant rules are loaded into the Workspace D/KB, the PCG of these rules shall be constructed and the cliques in the PCG identified.

- The cliques shall be ordered in such a way that a clique evaluation begins only after all its body predicates have been evaluated.

- The type of the each column of a base predicate is fixed at the time it is created. The type of the columns of the derived predicates shall be inferred from the rules. For example, in the rule $p(X, Y) - b(X, Y)$, the type of the first (respectively, second) column of p is the same as that of the first (respectively, second) column of b. Type checking shall infer the types of the derived predicates and also check whether the same types are inferred from all the rules defining p.

- The compiled query shall consist of either a standard relational query (when evaluating a non-recursive D/KB query) or an ordered list of LFP queries (one for each clique that is to be solved when evaluating a recursive D/KB query). An LFP query is a query that takes a set of recursive equations of the form, $r_i = f_i(r_1, ..., r_n)$, $i = 1, ..., n$, as input and computes their least fixed point, thereby solving each r_i. The f_i's are relational algebra expressions.

- Database (as opposed to rule base) queries shall be processed using traditional relational compilation and optimization techniques.

- A bottom-up strategy (see section 4.4) shall be adopted for rule base query processing. This is because bottom-up strategies are simpler and easy to implement. Bottom-up evaluation of a non-recursive predicate shall be done using traditional relational compilation and optimization techniques (see section 4.5). Bottom-up evaluation of a recursive predicate involves computing the LFP of a set of recursive equations. Policies for LFP evaluation are presented in section 8.7.

8.5. D/KB Update Processing Policies

This section presents policies that relate to updating the Stored D/KB with rules and facts from the Workspace D/KB.

- During updates, the D/KBMS shall ensure that after the update, *ireachablepreds* is the transitive closure of the PCG of the rules in *irulesource*.

- During updates, the D/KBMS shall ensure that the Workspace D/KB rules do not cause the type of any Stored D/KB derived predicate to change. If this is likely to happen, the update shall be rejected.

8.6. D/KBMS Functional Partitioning Policies

This set of policies relates to the functional components of a D/KBMS and their interfaces.

- The D/KBMS shall be partitioned into two layers, the *Knowledge Manager (KM)* and the *Data Base Management System (DBMS)*, with the KM at the top and the DBMS at the bottom. The KM provides the interface to the D/KBMS from the outside world.

  The basis for this partitioning is the division of D/KB query processing into compilation and execution phases. The KM and the DBMS correspond respectively to these phases: the KM is the D/KB query compiler, while the DBMS is the compiled query execution engine.

  The KM shall be responsible for query parsing, relevant rule extraction, magic set optimization, clique identification and ordering, and semantic checking. The DBMS shall be responsible for evaluating the cliques and non-recursive predicates as per the order prescribed by the KM.

- The KM/DBMS interface shall be relational algebra augmented with a general LFP operator and one or more specialized LFP operators. This interface determines the allocation of functions above and below the KM/DBMS layer boundary, and therefore, is a key D/KBMS design issue.

  The motivation for including relational algebra in this interface is the following. First, there is a close match between logic and relational algebra and this makes relational algebra an attractive starting point for Horn clause query processing support. Second, the set oriented nature of relational algebra makes powerful, nonprocedural query languages possible. Third, operations on sets and relations are inherently parallel.

  The reason for augmenting relational algebra with a general LFP operator is that it is not possible to express LFP queries (or recursive queries) using relational algebra alone. Such queries arise when a clique of mutually recursive predicates is to be evaluated. Since relational algebra cannot express LFP queries, cliques must be evaluated via an application program generated by the Knowledge Manager when using relational algebra as the KM/DBMS interface. There is not much scope for the KM to optimize the performance of this application program since the information needed for this optimization (join selectivities, intermediate relation sizes, etc.) is not visible to the KM through the relational algebra interface. On the other hand, suppose a general LFP operator is included in the KM/DBMS interface,

which accepts a set of recursive equations of the form, $r_i = f_i(r_1, ..., r_n)$, $i = 1, ..., n$, as input and computes their least fixed point, thereby solving each r_i. Then the KM need not generate an application program; it can simply generate LFP queries and the DBMS can figure out an efficient way to execute them (several policies for enhancing the performance of LFP evaluation are outlined in section 8.5).

The reason for including one or more specialized LFP operators in the KM/DBMS interface is that it may be possible to optimize the execution of certain special operators better than that of a general LFP operator. Since special LFP queries like transitive closure are expected to occur frequently, including such operators in the KM/DBMS interface and implementing them efficiently in the DBMS can enhance overall performance.

8.7. LFP Evaluation Policies

D/KB query execution performance is very significantly affected by the efficiency of the LFP evaluation strategy. This section presents policies relating to LFP evaluation.

- LFP evaluation shall be done using semi-naive evaluation (see algorithm 4 in chapter 4). The motivation for choosing semi-naive evaluation over naive evaluation is that it avoids much of the redundant work (i.e., recomputing tuples in an iteration that were computed in the previous one) of the latter.

- While simpler and easy to implement, bottom-up strategies compute a lot of useless results, since they do not use knowledge about the query to restrict the search space. To overcome this problem, the KM shall rewrite the relevant rules using the generalized magic set optimization algorithm (see section 4.4) into an equivalent set of rules whose bottom-up evaluation is more efficient. The policy of combining semi-naive evaluation with the generalized magic set optimization algorithm addresses the inefficiency problem of bottom-up strategies, while at the same time retaining their ease of implementation advantage.

- To enhance LFP evaluation performance, a dynamically adaptable indexing strategy shall be used to speed up the evaluation of the right hand side of the recursive equations or their differential. This strategy shall dynamically create and drop temporary indexes on the base and intermediate derived relations depending on their relative sizes.

- During LFP evaluation, the join strategy shall be dynamically changed between iterations if necessary, depending on the sized of the base and intermediate derived relations and the join selectivities from the previous iterations.

- Parallel and pipelined processing techniques shall be employed during LFP evaluation. These include evaluating the right hand side of each recursive equation in parallel and pipelining and data flow techniques for evaluating the relational algebra tree corresponding to the right hand side of these equations.

8.8. Join Processing Policies

The efficiency of the join operation has a significant impact on D/KB query execution performance. This is because during recursive query execution, evaluation of a clique generally involves numerous join operations. This section presents policies relating to join processing.

- The DBMS shall employ the parallel and pipelined join algorithms described in chapter 6.

- The query optimizer shall choose an appropriate algorithm for a particular join operation and system configuration. It shall carefully determine the number of clusters, the number of disks, and the number of processors that will be used in the join.

- In most cases, the query optimizer shall choose a hash-based join algorithm. This is because hash-based join algorithms are naturally parallelizable and indeed, the performance evaluation reported in chapter 8 confirms the relative superiority of hash-based join algorithms over sort-merge join algorithms. However, in cases where the output is required in sorted order or the source relations are already sorted, sort-merge based join algorithms shall be preferred.

8.9. D/KBMS Hardware Architectural Policies

This section presents policies relating to the D/KBMS hardware architecture.

- The KM and the DBMS shall be resident on different pieces of hardware, the KM on a general purpose workstation and the DBMS on a special purpose parallel relational database machine. The DBMS software and the parallel database machine shall constitute a data/knowledge base server. The overall environment shall be a local area network in which several workstations running the KM send LFP queries to this server.

- The DBMS shall be a multiprocessor relational database machine to get the requisite levels of performance. The DBMS architecture shall be parameterized in the degree of memory sharing, so that tightly coupled, loosely coupled, and intermediate architectures can be obtained. It shall use a large number (tens to hudreds, at least) of processors to obtain the necessary performance, a large amount (hundreds of megabytes to hundreds of gigabytes) of semiconductor memory, and shall support an aggregate disk capacity of a terabyte of more. It shall be constructed principally from commodity components such as general purpose microprocessors and conventional disk storage devices, since commodity components have superior price/performance and reliability compared to custom components.

- The architecture shall consist of a set of clusters linked by an intercluster bus or ring. Each cluster shall consist of a set of processors, a shared memory bank addressable by all the processors in the cluster, and a set of disk storage units and associated controllers. The processors may have local caches to reduce memory contention, but this shall be invisible to the data management software except for

possibly the need to flush the cache occasionally. Transfers between disk and memory, and between cluster memories over the bus, shall be a page at a time, where a page is a few kilobytes or more in size. A specific configuration of this architecture shall be determined by the following parameters: number of clusters, number of processors *per cluster*, number of disks *per cluster*, pages of main memory available *per cluster*, and page size in bytes. These range of values for these parameters shall be such that a wide range of performance and D/KB size requirements are accommodated.

- The effect of system configuration on join performance is significant. A configuration with a few large clusters shall be preferred to one with many small clusters. In fact, our investigation showed that a very large single cluster provides the best performance, since the communication cost is eliminated.

- A system with many clusters shall be preferred to one with fewer clusters of the same size. This variation increases the parallel processing power of the system and speeds up the data transfer rate for some algorithms because the data is distributed better in the system.

- The number of disks at each cluster shall be increased if the disk I/O proves to be a bottleneck.

- Except for systems employing processing intensive join algorithms, the number of processors at each cluser shall not be increased as a means of enhancing performance. This is because the CPU processing speed is not the bottleneck in most cases.

- The effect of disk I/O speed and network transfer rates on join algorithm performance is significant. Therefore, the values of these components shall be adjusted to achieve the required levels of cost/performance. If disk I/O still becomes a bottleneck, other solutions such as, using a large page size, using a large disk-memory transfer rate, increasing the number of disks at each cluster, and increasing number of clusters, shall be employed.

8.10. Fault Tolerance Policies

Fault tolerance is a system's ability to tolerate faults in its components. Fault tolerance techniques make a system tolerate faults, and in case of failures that cannot be tolerated, allow the system to degrade gracefully (i.e., allow partial operation). The parameter most often used to measure the fault tolerance of a system is its availability, which is the percentage of time a system performs according to its specifications (i.e., is available to do useful work). This section presents policies for ensuring high D/KBMS availability in very large D/KB environments.

- The D/KBMS shall employ the hardware fault tolerance techniques described in section 7.3.2 to tolerate failures of D/KBMS hardware components affecting availability. These components are: disks, processors, memory, interconnect, and power supply. Failure of a sixth component, the intra-cluster bus, is very infrequent and can be considered as manifesting itself as a failure of one of the other five

components. The hardware fault tolerance techniques introduce redundancy at the component level so that component failures can be tolerated without affecting system operation.

- The D/KBMS shall employ the software fault tolerance techniques described in section 7.3.3 to tolerate failures affecting the major software components of the D/KBMS. These components are: transaction management and operating system software.

- From a fault tolerance point of view, architectures with small clusters (i.e., less than 8 disks per cluster) and a low degree of memory sharing shall be preferred to architectures with large clusters and a high degree of memory sharing, as the former have better fault tolerance. However, exactly the opposite is true from the query processing point of view. The choice depends on the application requirements. If fault tolerance is more important, a large number of small clusters shall be used, while if query processing performance is more important, a few large clusters shall be used.

- The D/KBMS shall employ components with high individual reliability values (particularly disks) and high capacity (particularly disks and interconnect) as the Phase I investigation showed that such components have a significant positive effect on D/KBMS availability.

- The D/KBMS shall employ data storage methods with high replication and high distribution factors.

- The D/KBMS shall employ access methods that allow large amount of parallelism Larger page sizes shall be preferred in a D/KBMS with many small clusters.

- Query processing and optimization techniques and system fault tolerance are very intimately related. A comparison of the results of our parallel join processing and fault tolerance investigations present interesting but difficult trade-offs. The join processing investigation shows that algorithms perform better in a tightly coupled (single cluster) architecture since the communications cost is eliminated. However, the fault tolerance of such an architecture is the worst since failure of one component unit, such as the shared memory, can stop the complete system. In another example, consider the trade-offs with respect to the data storage scheme. The query processing techniques and the data storage schemes depend on the type and frequency of queries asked (i.e., the application). As we saw in our investigation, the data storage scheme affects the fault tolerance significantly. Thus, in designing a real system, the designers of query processing software and fault tolerance have to understand the application and devise a storage scheme that is acceptable from the query processing as well as fault tolerance viewpoint. Required levels of response time, fault tolerance and reliability are application dependent.

8.11. Steps in D/KB Query and Update Processing

We first describe the steps for processing queries to the workspace D/KB, for the case where the rules in the workspace D/KB do not refer to rules in the stored D/KB

(scenario 1). We then describe how to update the stored D/KB with rules and facts from the workspace D/KB. Finally, we describe workspace and stored D/KB query processing (scenario 2) for the case where the rules in the workspace D/KB and the stored D/KB may refer to each other.

8.11.1. Scenario 1

Consider the following rules and query.

R_1: $ancestor(X, Y) - parent(X, Y)$.

R_2: $ancestor(X, Y) - parent(X, Z), ancestor(Z, Y)$.

$$query(X) - ancestor("john", X).$$

1. Add the query rule to the predicate connection graph. The PCG then is as shown in figure 8.1.
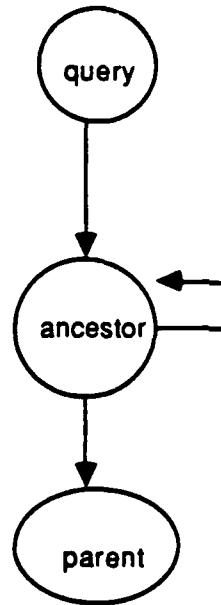
Figure 8.1. PCG with Query Rule Added

2. Find relevant predicates and rules. The relevant predicates are the predicate reachable from the query node in the PCG. The relevant predicates can be obtained by computing the transitive closure of the PCG. For the *ancestor* query, the relevant predicates are *ancestor* and *parent*. The relevant rules are those defining derived relevant predicates. For the *ancestor* query, the relevant rules are R_1 and R_2. We also add *query* to the set of relevant predicates and the query rule to the set of relevant rules.

3. Generate the adorned versions of the relevant predicates and adorned rules defining these predicates.

$$R_1: \quad ancestor^{bf}(X,\ Y) - parent(X,\ Y).$$

$$R_2: \quad ancestor^{bf}(X,\ Y) - parent(X,\ Z),\ ancestor^{bf}(Z,\ Y).$$

$$query^f(X) - ancestor^{bf}("john",\ X).$$

4. Generate magic rules and modified rules.

$$R_1: \quad magic\_ancestor^{bf}(Z) - magic\_ancestor^{bf}(X),\ parent(X,\ Z).$$

$$R_2: \quad magic\_ancestor^{bf}("john").$$

$$R_3: \quad ancestor^{bf}(X,\ Y) - magic\_ancestor^{bf}(X),\ parent(X,\ Y).$$

$$R_4: \quad ancestor^{bf}(X,\ Y) - magic\_ancestor^{bf}(X),\ parent(X,\ Y),\ ancestor^{bf}(Z,\ Y).$$

$$query^f(X) - ancestor^{bf}("john",\ X).$$

5. Split the data/knowledge base into its intensional and extensional components.

$$R_1: \quad magic\_ancestor^{bf}(Z) - magic\_ancestor^{bf}(X),\ parent(X,\ Z).$$

$$R_2: \quad magic\_ancestor^{bf}(X) - base\_ma^{bf}(X).$$

$$R_3: \quad ancestor^{bf}(X,\ Y) - magic\_ancestor^{bf}(X),\ parent(X,\ Y).$$

$$R_4: \quad ancestor^{bf}(X,\ Y) - magic\_ancestor^{bf}(X),\ parent(X,\ Y),\ ancestor^{bf}(Z,\ Y).$$

$$base\_ma^{bf}("john").$$

6. Construct the PCG of these rules.

7. Find the cliques of this PCG (see figure 8.2).

8. Construct the *evaluation graph*. This is a directed graph whose nodes are either derived predicates or cliques. There can be four types of directed edges: (1) $P \to C$ indicates that some predicate in the clique C appears in the body of a rule defining P, (2) $C \to P$ indicates that P appears in the body of a rule defining some predicate of C, (3) $P_1 \to P_2$ indicates that P_2 appears in the body of a rule defining P_1, and (4) $C_1 \to C_2$ indicates that some predicate of C_2 appears in the body of a rule defining some predicate of C_1.

For the *ancestor* query, the evaluation graph is shown in figure 8.3. The evaluation graph is essentially the PCG with the base predicates removed and the

	Recursive predicates	$ancestor^{bf}$
Clique C1	Recursive rules	R_4
	Exit rules	R_3
	Recursive predicates	$magic\_ancestor^{bf}$
Clique C2	Recursive rules	R_1
	Exit rules	R_2

Figure 8.2: Cliques for ancestor query
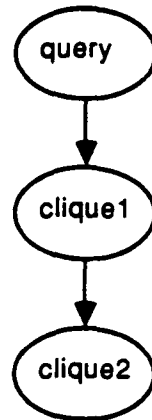
Figure 8.3. Evaluation Graph for ancestor Query

predicates of a clique collapsed into a single node. Thus, the predicate nodes of the evaluation graph are the nonrecursive predicates of the PCG. Also, it should be clear that while the PCG may be cyclic, the evaluation graph is acyclic.

9. Perform a topological sort of the evaluation graph. This gives a total order indicating the order in which the nonrecursive predicates and cliques are to be evaluated. The order is such that a rule is evaluated only after the predicates in the body are evaluated. The total order is called the *evaluation order list*.

For the *ancestor* query, the total order is C2, C1, *query*.

10. Perform semantic checks. In this work, we consider two kinds of checks. The first is to check for each derived relevant predicate whether there is a rule defining it. The second is a type check. The type of each column of a base predicate is fixed at the time it is created. The type of the columns of the derived predicates is inferred from the rules. For example, in the rule $p(X, Y) - b(X, Y)$, the type of the first (respectively, second) column of p is the same as that of the first (respectively, second) column of b. Type checking involves inferring the types of the derived predicates and also checking whether the same types are inferred from all the rules defining p. This is easy to do for nonrecursive predicates. However, for

recursive predicates, we need to loop till either closure is reached or there is a type mismatch. We have developed a type checking algorithm, which we describe in section 8.11.1.1.

If there is an error in either of the two semantic checks, we do not perform the next step.

11. Evaluate the cliques and nonrecursive predicates as per the order of step 9. Evaluating a clique means evaluating a block of mutually recursive predicates. This is done as described in section 4.6. Nonrecursive predicate evaluation is done as described in section 4.5. []

8.11.1.1. Type Checking Algorithm

Type checking in data/knowledge base processing has two purposes: the first purpose is to determine the types of each of the derived predicates. The second purpose is to ensure that the type of each respective column of a predicate is the same in every occurrence of the predicate, which is similar to type checking by a compiler of any strongly typed language.

The type of each column of a base predicate is fixed at the time it is created. During the type checking, the type of each column of a base predicate is available from the data dictionary of the database that stores the relations corresponding to the base predicates. The type of a column of a derived predicate is not known before the type checking is performed. It is inferred from relationships in the rules using the type of columns of base predicates and the type of columns of the derived predicates which are already inferred.

Example:

$R1: p(X,Y) \leftarrow b1(X,Y).$

Here the type of the first (respectively, second) column of p is the same as that of the first (respectively, second) column of $b1$.

$R2: p(X,Y) \leftarrow b2(X,Z), q(Z,Y).$

Here type of the first column of p is the same as that of the first column of $b2$ and the type of the second column of p is the same as the type of the second column of q. Additionally, the type of the first column of q is the same as the type of second column of $b2$. []

If the rule set contains both rules, the type of columns of p inferred using both rules should be the same. Thus, in the above example, if type of the first column of $b1$ is not the same as that of the first column of $b2$ then the type checking will give error since different types for the first column of p will be inferred using the two rules. Similarly, if the type of the second column of $b1$ is not the same as that of the second column of q, the type checking will given an error since different types of the second column of p will be inferred using the two rules.

Type checking is easy to do for a nonrecursive predicate. However, for recursive predicates, it may involve several iterations until either a closure is reached or a type

mismatch is found.

Example:

        R1: p(X,Y) ~ b1(X,Y).
        R2: p(X,Y) ~ b2(X,Z), q(Z,Y).
        R3: q(X,Y) ~ p(X,Z), b3(Z,Y).

If the rules are evaluated in order *R1*, *R2* and *R3*, the type of columns of predicate *q* will not be known when *R2* is evaluated in the first iteration. However after evaluating *R3*, the type of columns of *q* will be known and hence the type check can be completed by evaluating *R2* again. []

Type check for a recursive predicate is performed by evaluating together all the rules in the cliques to which it belongs. However, before a clique can be evaluated, types of all the nonrecursive predicates should be known. While the types of nonrecursive predicates that are base predicates are known from the data dictionary of the database, types of other nonrecursive derived predicates can be known if type checking follows the total order defined by the evaluation order list. All exit rules in a clique are evaluated before evaluating the recursive rules.

The important steps of the type checking algorithm are as follows.

1.   Evaluate for type checking the nodes of the evaluation graph according to the total order (see steps 8 and 9 of section 8.11.1).

2.   If the node is a clique, evaluate for type checking the exit rules first, followed by the recursive rules. Iterate through the recursive rules until type of all predicates are known, and the type of columns derived in every occurrence of a predicate is found to be consistent or a type mismatch is found.

Example: Consider rules given in figure 4.1 and the corresponding cliques shown in figure 4.3. According to the total order, Clique 2 (or Clique 3) will be evaluated first, followed by Clique 3 (or Clique 2), and then Clique 1. Before Clique 3 is evaluated, the type of all derived nonrecursive predicates in Cliques 3 (*p1* and *p2*, in this case) will be known. First the exit rule, *R2*, will be evaluated, followed by the recursive rules, *R1* and *R7*. []

8.11.2. Updates to the Stored Data/Knowledge Base

In this section, we describe our algorithm for updating the stored D/KB with rules and facts from the workspace D/KB. In this work, we only consider the storage structures for storing rules and facts on disk. We do not consider checking the workspace D/KB rules and facts against integrity constraints that may be associated with the stored D/KB.

The storage structures for rules and the update algorithm described in this section are our work; they do not appear in the literature.

The storage structures for base predicates are the usual relations of relational algebra. Each base predicate is a table in the relational database. The data dictionary of the relational database is used to store information about the columns of base predicates.

We propose four relations as the storage structures for rules. These relations are called *isystables*, *isyscolumns*, *irulesource*, *ireachablepreds*. *isystables* and *isyscolumns* are the data dictionary of the intensional knowledge base. They contain the types of the columns of the derived predicates. These types are inferred using the type checking algorithm described in the last section.

The rule storage structures have the following schema:

> *isystables*(*tablename char, tableid integer*)

> *isyscolumns*(*tableid integer, colname char, colnumber integer, coltype integer*)

irulesource stores for each derived predicate p, the rules defining p. It has the following schema:

> *irulesource*(*headpredname char, rule char*)

ireachablepreds is the transitive closure of the PCG of the rules stored in *irulesource*. It stores for each derived predicate p all the predicates reachable from p. It has the following schema:

> *ireachablepreds*(*frompredname char, topredname char*).

We now illustrate how the *ancestor* rules would be stored using the above storage structures. Assume that both columns of the *parent* relation are of type char(30). Then the type checking algorithm will infer that both columns of *ancestor* are also of type char(30). We add tuples to the storage structures as follows.

> *isystables*(*ancestor*, t_{anc})

> *isyscolumns*(t_{anc}, *ancestor_col_1*, 1, *char*(30))
> *isyscolumns*(t_{anc}, *ancestor_col_2*, 2, *char*(30))

> *irulesource*(*ancestor*,"*ancestor*(X, Y) -- *parent*(X, Y).")
> *irulesource*(*ancestor*,"*ancestor*(X, Y) -- *parent*(X, Z), *ancestor*(Z, Y).")

> *ireachablepreds*(*ancestor*, *parent*)
> *ireachablepreds*(*ancestor*, *ancestor*)

The major advantage of *ireachablepreds* as a compiled form of the rules is that it allows very efficient retrieval of the relevant rules from the intensional knowledge base. For example, suppose the stored D/KB contains the following rules:

> R_1: $p(X, Y)$ -- $a(X, Z)$, $q(Z, Y)$.

R_2: $a(X, Y) \leftarrow b_1(X, Z), a(Z, Y)$.

R_3: $a(X, Y) \leftarrow b_2(X, Y)$.

R_4: $q(X, Y) \leftarrow c(X, Y)$.

R_5: $c(X, Y) \leftarrow b_3(X, Y)$.

R_6: $m(X, Y) \leftarrow b_4(X, Y)$.

where the b_i's are base predicates. Then retrieving all the rules needed to solve the query

$query(X, Y) \leftarrow p(X, Z), m(Z, Y)$.

is accomplished via the following SQL query:

SELECT irulesource.rule

FROM irulesource, ireachablepreds

WHERE (ireachablepreds.topredname = irulesource.headpredname OR

        ireachablepreds.frompredname = irulesource.headpredname) AND

        ireachablepreds.frompredname = "p" OR

        ireachablepreds.frompredname = "m"))

This query retrieves rules R_1 through R_6 above. To speed up the execution of this query, we place a composite index on the columns of *ireachablepreds*.

We now describe the update algorithm. Let ΔDKB denote the workspace D/KB.

1. Extract from the stored D/KB all the rules needed to evaluate the derived predicates in ΔDKB. This can be done using a query like the above SQL query. Let IKB_{rel} denote the extracted rules.

2. Construct the PCG of the rules in $\Delta DKB_{composite} = \Delta DKB \cup IKB_{rel}$, which denotes the set of rules which are either in ΔDKB or in IKB_{rel}.

3. Compute the transitive closure of this PCG. This gives all the predicates reachable from a given predicate in $\Delta DKB_{composite}$.

4. Perform the two semantic checks described in step 10 of scenario 1.

5. For each derived predicate p in $\Delta DKB_{composite}$, add tuples to *isystables* and *isyscolumns* if information on p is not present in these tables.

6. For each derived predicate p in $\Delta DKB_{composite}$ add tuples to *ireachablepreds* by looking at the transitive closure computed in step 3.

7. For each rule in ΔDKB, add tuples to *irulesource*. []

We mentioned before that *ireachablepreds* is the transitive closure of the PCG of the rules in *irulesource*. The above algorithm computes this transitive closure incrementally. That is, whenever the stored D/KB is to be updated, we do a transitive closure on only those portions of the stored D/KB that will be affected by the update (IKB_{rel}), and not of the entire stored D/KB. This can result in substantial savings in update times for very large rule sets as the size of IKB_{rel} will be much smaller than that of the entire rule set.

8.11.3. Scenario 2

In this section, we describe workspace and stored D/KB query processing for the case where the rules in the workspace D/KB and the stored D/KB may refer to each other. In this case, we need to extract all the relevant rules from the stored D/KB. We replace step 3 of scenario 1 with the following steps.

3.1. Compute the transitive closure of ΔDKB.

3.2. From the transitive closure, find the predicates reachable from the query. Let P denote this set of predicates.

3.3. Extract from the stored D/KB all the rules needed to evaluate the predicates in P. Let IKB_{rel} denote the extracted rules.

4. Compute the transitive closure of $\Delta DKB_{composite} = \Delta DKB \cup IKB_{rel}$. This gives the correct set of relevant predicates and rules. []

We point out that *ireachablepreds* makes it possible to efficiently extract the relevant rules from the stored D/KB. This efficiency will translate to higher performance, especially for very large rule sets.

8.12. Steps in D/KBMS Architecture Specification

This section presents an overall procedure for specifying D/KBMS architectures. The steps below are not intended to be comprehensive. Where gaps exist, the D/KBMS designer should consult the policies described earlier in this chapter to determine an appropriate course of action.

●   Design a high performance parallel relational database machine that employs the parallel and pipelined join algorithms described in chapter 6.

●   Implement the hardware and software fault tolerance techniques described in chapter 7.

●   Design parallel algorithms for general LFP evaluation using the above join strategies, data flow and pipelining techniques, and semi-naive evaluation.

●   Design parallel algorithms for special LFP operators such as transitive closure using the HYBRIDTC strategy outlined in chapter 5.

●   Design a Knowledge Manager that compiles Horn clause queries to relational algebra augmented with a general LFP operator and that uses the generalized magic

sets strategy for restricting the search space to the relevant base relation tuples. The Knowledge Manager should follow the steps for D/KB query and update processing described in the previous section.

8.13. Conclusions

This chapter presented a methodology for specifying high performance, highly available, large D/KBMS architectures. The methodology was described as a set of policies and steps. The policies are meant to serve as a guide to the D/KBMS designer in making appropriate decisions for the following critical D/KBMS design issues: overall D/KBMS functionality, knowledge representation, rule storage, D/KB query processing, D/KB update processing, D/KBMS functional partitioning, LFP evaluation, join processing, D/KBMS hardware architecture, and fault tolerance. The steps were presented as a recipe for D/KB query and update processing and for D/KBMS architecture specification.

CHAPTER 9

VLPDF Demonstration Testbed

This chapter describes a data/knowledge base management testbed that we have designed and implemented on top of a commercial relational database system. The testbed is intended to serve as both a demonstration and performance measurement and evaluation platform. As a demonstration platform, the testbed illustrates the motivation and basic functionality of a D/KBMS, the components of a D/KBMS architecture, alternative implementations of these components and their relative tradeoffs, and the factors contributing to D/KB query compilation and execution time. As a performance measurement and evaluation platform, the testbed allows us to make quantitative performance measurements and to study system performance sensitivity and behavior with respect to several parameters.

9.1. VLPDF Demonstration Testbed Architecture

The VLPDF demonstration testbed is built on top of an existing testbed — the Informix Relational Database Management System (RDBMS) — and is implemented in a Unix 4.2 BSD environment on Apollo workstations. The overall configuration of the testbed is shown in Figure 9.1. The testbed consists of four components: User Interface, Knowledge Manager, Informix Relational DBMS, and C Compiler.

The Knowledge Manager together with the Informix RDBMS constitutes the D/KBMS. The User Interface manages the interaction between users and the D/KBMS. The users can be either humans or application programs (we also view expert system as application programs).

The Knowledge Manager is essentially a compiler. It accepts Horn clauses and queries from the User Interface and compiles queries into C code fragments. The C code fragment is then compiled by the C compiler and linked with a run-time library to produce the object code, which is executed by the User Interface as an application program against the Informix RDBMS to give the query results. The C code fragment contains information specific to the query, while the run-time library contains the algorithms for LFP evaluation and miscellaneous utilities.

9.1.1. User Interface

The User Interface provides the following options:

Quit

This option terminates the current session with the demonstration testbed.

SQL
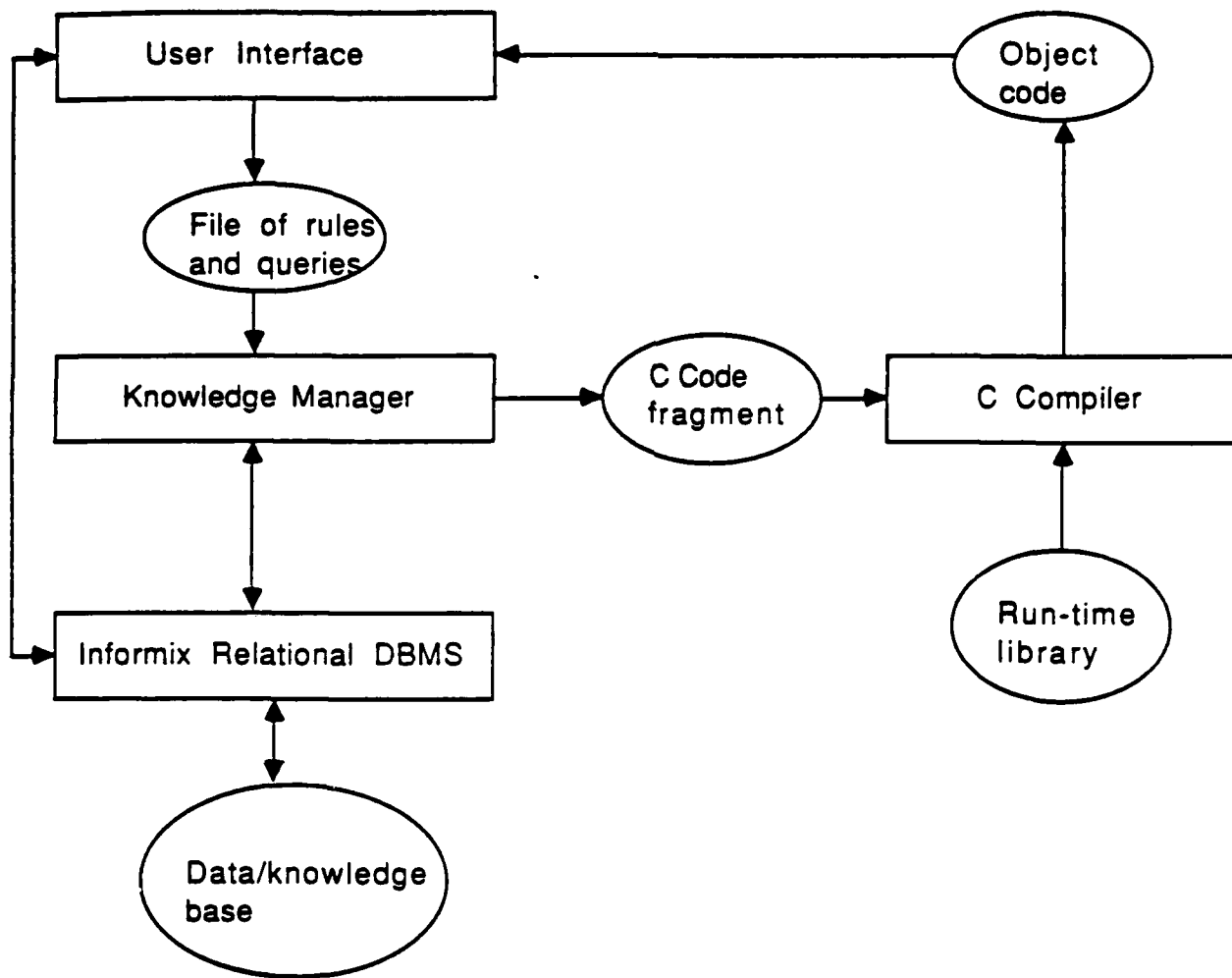
Figure 9.1. VLPDF Demonstration Testbed

This option allows the user to directly access the Informix RDBMS through an SQL interface.

Embedded Horn Clauses/PARLOG

PARLOG is a language with two types of relations: single-solution relations and all-solutions relations. All-solutions relations can appear in the body of single-solution relations. The evaluation semantics for these relations are completely different. Indeed, they can be viewed as two different languages — all-solutions relations constituting a query language and single-solution relations a parallel applications programming language. An all-solutions relation query can be evaluated by computing the least fixed point of the Horn clauses defining this relation. Thus, PARLOG can be considered as

having a Horn clause query language embedded within it.

This option allows the user to enter and query single-solution PARLOG clauses. These clauses may contain calls to all-solutions relations in their body. The Horn clauses defining these all-solutions relations are assumed to be in the stored D/KB.

D/KBMS Interaction

This option allows the user to interact with the D/KBMS. The D/KBMS interaction options constitute the second level menu and are described below.

Set D/KB

This option allows the user to specify the name of the stored D/KB against which to process queries and updates.

Enter Rules

This option allows the user to enter a set of Horn clauses into the workspace D/KB, either interactively or through a file.

Enter Query

This option allows the user to enter a query. The entered query will get compiled by the Knowledge Manager. The User Interface prompts the user for the query and for the name of the object file in which to put the compiled query. It then puts the entered query into a file and requests the Knowledge Manager to compile the query.

Execute Query

This option allows the user to execute a previously entered query. The User Interface executes the previously compiled query as an application program against the Informix RDBMS.

Update Stored D/KB

This option allows the user to update the stored D/KB with rules and facts from the workspace D/KB.

Quit

This option allows the user to return to the top-level menu.

Initialize D/KBMS

This option allows the user to initialize the major data structures of the Knowledge Manager.

List Workspace D/KB Rules

This option allows the user to view the rules present in the workspace D/KB.

List Relevant Stored D/KB Rules

This option allows the user to view all the rules present in the stored D/KB required to evaluate a given predicate. The User Interface prompts the user for the predicate name and arity. It then requests the Knowledge Manager to list the stored rules needed to evaluate this predicate, giving it

the predicate name, the arity, and the stored D/KB name. The Knowledge Manager extracts the relevant rules from the storage structures and puts them in a file, which is then displayed by the User Interface.

9.1.2. Knowledge Manager

In this section, we describe the architecture of the Knowledge Manager. The Knowledge Manager consists of the following components: Rule Parser, Stored D/KB Manager, Workspace D/KB Manager, Semantic Checker, and Code Generator. The architecture of the Knowledge Manager, indicating the interconnections between these components, is shown in figure 9.2. The circles in this figure represent data structures and the boxes, components.

First, we list the functions provided in the Knowledge Manager interface and the functions provided in the various components' interface. Second, we describe the major data structures of the Knowledge Manager. Finally, we describe the processing done by the Knowledge Manager to implement the functions in its interface.

9.1.2.1. Knowledge Manager Interfaces

Overall Knowledge Manager Functions

```
Initialize D/KBMS

Enter Horn clauses

Compile query

Update stored D/KB

List workspace D/KB rules

List relevant stored D/KB rules
```

Workspace D/KB Manager Interface

```
Compute PCG transitive closure

Extract relevant predicates

Find cliques

Generate evaluation order list
```
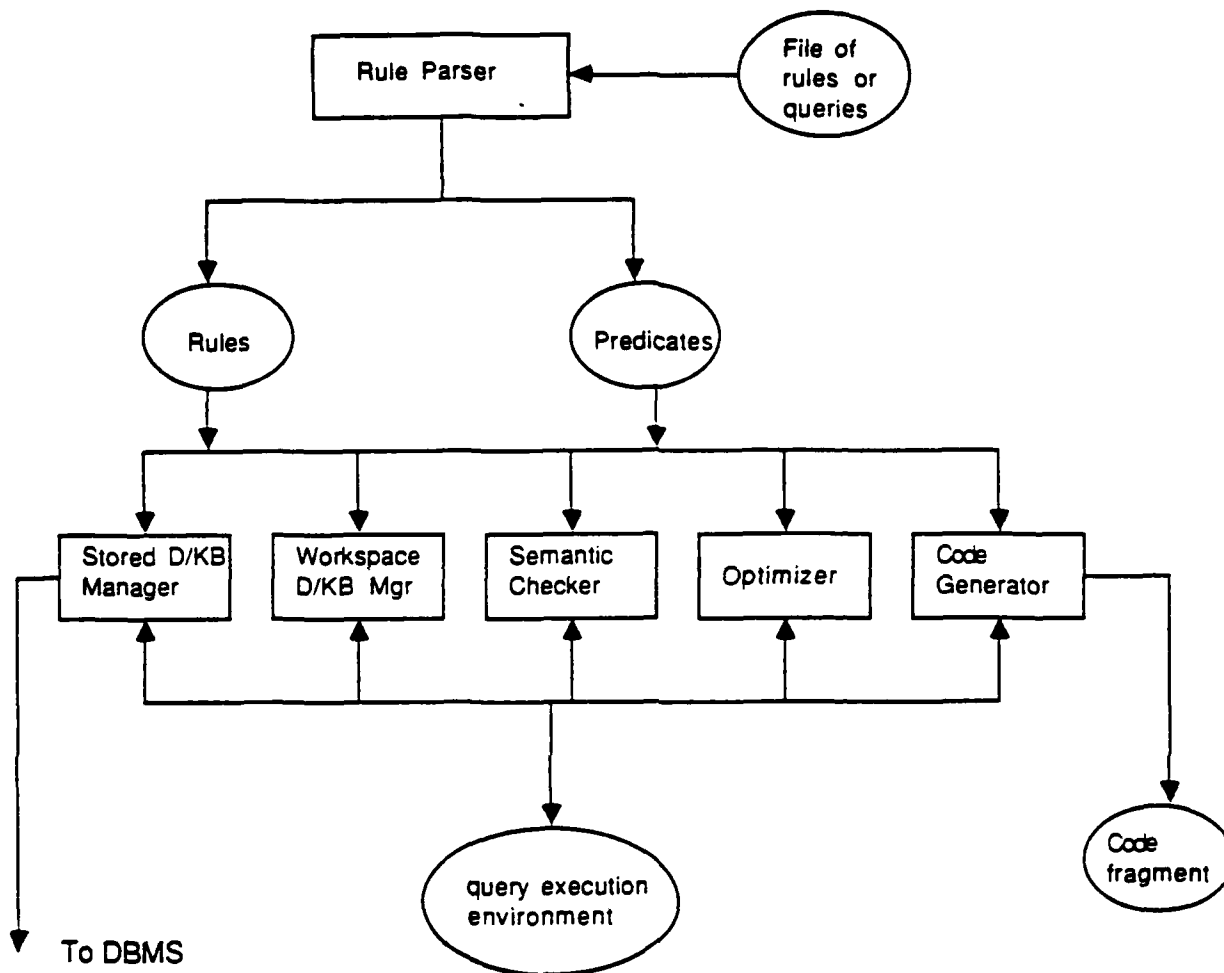
Figure 9.2. Knowledge Manager Architecture

Stored D/KB Manager Interface

Extract relevant rules

Insert temporary base predicates

Read data dictionary

Update D/KB

Optimizer Interface

Generate adorned rule set
Generate magic and modified rules

9.1.2.2. Knowledge Manager Data Structures

The major data structures in the Knowledge Manager are: *rules*, *predicates*, and *query execution environment.*

Rules

This a hash table with each bucket containing the following information:

- Rule id.
- Name and list of arguments of each predicate as indicated in the source form of the rule.
- Other information about each predicate (see below).

Predicates

This a hash table with each bucket containing the following information:

- Predicate id.
- Internal name.
- Type, i.e., base or derived.
- Arity (number of arguments).
- Schema information, i.e., name and type of each column.
- List of rules for which the predicate is a head.
- List of rules for which the predicate is a body predicate.

Query execution environment

The query execution environment data structure contains all the information necessary to execute a query to the data/knowledge base.

- *tcpcg*: transitive closure of the PCG of the rules in the workspace D/KB (the transitive closure graph is represented as a boolean matrix).
- *relpreds*: list of predicates and rules relevant to the query.
- *cliques*: list of cliques in the workspace D/KB. Each clique contains:
    - Clique id.
    - List of exit rules.
    - List of recursive rules.
    - List of recursive predicates.

- *evalorderlist*: list of nodes of the evaluation order graph for the query in topologically sorted order.

9.1.2.3. Knowledge Manager Processing

In this section, we describe the processing done by the Knowledge Manager to implement the functions provided by its interface.

Initialize D/KBMS

The Knowledge Manager sets the *rules* and *predicates* hashtables to null.

Enter Horn Clauses

The Knowledge Manager asks the Rule Parser to parse the Horn clauses. The Rule Parser loads the *rules* and *predicates* hash tables.

Compile Query

1. The Knowledge Manager constructs a *query execution environment* data structure. It then asks the Workspace D/KB Manager to extract the predicates relevant to the query from the workspace D/KB. These are the predicates reachable from the query. The Workspace D/KB Manager computes the transitive closure of the PCG of the workspace D/KB rules to determine the reachable predicates. It fills the *tcpcg* and *relpreds* fields of the *query execution environment*.

2. The Knowledge Manager asks the Stored D/KB Manager to extract from the stored D/KB all the rules needed to evaluate the relevant predicates found in the previous step. The Stored D/KB Manager extracts the relevant rules and calls the Rule Parser to load them into the *rules* and *predicates* hash tables. The extracted rules then become part of the workspace D/KB.

3. The Knowledge Manager asks the Workspace D/KB Manager to extract the relevant predicates from the workspace D/KB.

4. The Knowledge Manager asks the Optimizer to rewrite the workspace D/KB rules into a more efficiently executable form. First, the Optimizer generates adorned versions of the relevant predicates and adorned rules defining these predicates, including the adorned version of the query (the result of evaluating the adorned query against the adorned rule set is the same as the result of evaluating the

original query against the original rule set). Second, it calls the Rule Parser to load the adorned rules and predicates into the *rules* and *predicates* hash tables. Third, it generates the magic and modified rules. Fourth, it calls the Rule Parser to load them into the *rules* and *predicates* hash tables. Finally, the Optimizer asks the Stored D/KB Manager to insert the relevant base predicates present in the workspace D/KB into the stored D/KB.

5. The Knowledge Manager sets the *tcpcg* and *relpreds* fields of the *query execution environment* to null. It then asks the Workspace D/KB Manager to extract the predicates relevant to the adorned query from the workspace D/KB. These are the predicates reachable from the adorned query. The Workspace D/KB Manager fills the *tcpcg* and *relpreds* fields of the *query execution environment.*

6. The Knowledge Manager asks the Workspace D/KB Manager to find the cliques in the workspace D/KB. The Workspace D/KB Manager fills the *cliques* field of the *query execution environment.* It assigns an id to each clique and fills in the exit rules, recursive rules, and recursive predicates for it.

7. The Knowledge Manager asks the Workspace D/KB Manager to generate the evaluation order list. The evaluation order list is a topological sort of the evaluation graph of the adorned query. The Workspace D/KB Manager fills the *evalorderlist* field of the *query execution environment.*

8. The Knowledge Manager asks the Semantic Checker to perform semantic checks. The Semantic Checker performs two types of checks. The first is to check for each derived relevant predicate whether there is a rule in the workspace D/KB defining it. The second is a type check. The Semantic Checker asks the Stored D/KB Manager to read the extensional and intensional data dictionaries to get schema information from the stored D/KB. See section 2.8 for the details of the type check algorithm. If the Semantic Checker reports no errors, the Knowledge Manager performs the next step.

9. The Knowledge Manager asks the Code Generator to generate the code for each entry in the *evalorderlist.* The Code Generator generates a C code fragment containing information specific to the query.

Update Stored D/KB

1. The Knowledge Manager constructs a *query execution environment* data structure. It then asks the Stored D/KB Manager to extract from the stored D/KB all the rules needed to evaluate the predicates in the workspace D/KB. The Stored D/KB Manager extracts the relevant rules and calls the Rule Parser to load them into the *rules* and *predicates* hash tables. The extracted rules then become part of the workspace D/KB.

2. The Knowledge Manager asks the Workspace D/KB Manager to compute the transitive closure of the PCG of the rules in the workspace D/KB. The Workspace D/KB Manager fills the *tcpcg* field of the *query execution environment.*

3. The Knowledge Manager sets the *relpreds* field of the *query execution environment* to the list of all predicates in the workspace D/KB.

4. The Knowledge Manager asks the Workspace D/KB Manager to find the cliques in the workspace D/KB. The Workspace D/KB Manager fills the *cliques* field of the *query execution environment.*

5. The Knowledge Manager asks the Workspace D/KB Manager to generate the evaluation order list. The Workspace D/KB Manager fills the *evalorderlist* field of the *query execution environment.*

6. The Knowledge Manager asks the Semantic Checker to perform the two semantic checks mentioned above. If the Semantic Checker reports no errors, the Knowledge Manager performs the next step.

7. The Knowledge Manager asks the Stored D/KB Manager to update the stored D/KB. The Stored D/KB Manager updates the intensional data dictionary and stores the compiled and source forms of the workspace D/KB rules.

List relevant stored D/KB rules

The Knowledge Manager asks the Stored D/KB Manager to extract from the stored D/KB all the rules needed to evaluate the indicated predicate. It then puts these rules in a file and asks the User Interface to display it.

9.1.3. Compiled Code and Run-time Library

The Knowledge Manager compiles data/knowledge base queries into C code fragments. The C compiler compiles a fragment and links it with the run-time library to produce an object program, which when executed against the Informix RDBMS gives the query results. In this section, we describe the contents of the C code fragment and the run-time library.

C code fragment

The C code fragment generated by the Knowledge Manager basically loads certain data structures in the object program with information specific to the query. These data structures contain information similar to the nodes of the evaluation order graph of the query. Recall that this graph contains two types of nodes — predicates and cliques. The predicate nodes contain information about evaluating non-recursive predicates, while clique nodes contain information about evaluating a block of mutually recursive predicates.

For predicate nodes, the C code fragment loads the predicate name, schema information (name and type of each column), and the SQL query to evaluate the body of each rule in which the predicate appears as head (see section 2.5 for what this SQL query looks like). For clique nodes, the C code fragment loads the same information, except that it differentiates between exit rules and recursive rules.

Run-time library

The run-time library contains routines that interpret the information loaded by the C code fragment. Non-recursive predicates are evaluated as described in section 2.5, while recursive predicates as in section 2.6. The major contents of the run-time library are the embedded SQL routines for naive and semi-naive evaluation of a system of

recursive equations. We describe these below.

```
naive_evaluation (clique)

/* Embedded SQL algorithm for naive evaluation of a clique. */

{
        changed = TRUE;

        /* Initialization */
        foreach predicate p in the clique {

                /* Create temporary tables */
                create table delta_p;
                create table p;
                create table temp_p;
                create table exit_p;

                /* Evaluate exit rules */
                insert into p tuples resulting from evaluating the SQL query associated
                with each exit rule for which p is the head;

                copy p into exit_p;
        }

        /* While loop */
        while (changed) {
                changed = FALSE;

                foreach predicate p in the clique {
                        copy exit_p into delta_p;

                        /* Evaluate right hand side of recursive equation */
                        insert into delta_p tuples resulting from evaluating the SQL query
                        associated with each recursive rule for which p is the head;

                }

                /* Termination check */
                foreach predicate p in the clique {
                        temp_p = delta_p - p;

                        If there are tuples in temp_p
                                changed = TRUE;

                        delete all tuples from temp_p;
                }

                foreach predicate p in the clique {
                        drop table p;
                        rename table delta_p to p;
                        create table delta_p;
                }
        }

        /* Clean up */
        foreach predicate p in the clique {
                drop table delta_p,
                drop table temp_p;
                drop table exit_p;
        }
}

seminaive_evaluation (clique)
```

```
/* Embedded SQL algorithm for semi-naive evaluation of a clique. */

{
        changed = TRUE;

        /* Initialization */
        foreach predicate p in the clique {

                /* Create temporary tables */
                create table p;
                create table new_p;
                create table DELTA_p;
                create table delta_p;

                /* Evaluate exit rules */
                insert into delta_p tuples resulting from evaluating the SQL
                query associated with each exit rule for which p is the head;

                copy delta_p to new_p;
        }

        /* While loop */
        while (changed) {

                foreach predicate p in the clique {
                        delete all tuples from DELTA_p;

                        /* Evaluate right hand side of recursive equations */
                        insert into DELTA_p tuples resulting from evaluating the
                        differential of each recursive rule for which p is the head;
                }

                foreach predicate p in the clique {
                        copy delta_p to p;
                        delete all tuples from delta_p;

                        /* Set difference operation for termination check */
                        delta_p = DELTA_p - p;
                }

                foreach predicate p in the clique {
                        copy delta_p to NEW_p;

                        If there are tuples in NEW_p
                                changed = TRUE;
                }
        }

        /* Wrap up */
        foreach predicate p in the clique {
                drop table p;
                rename table new_p to p;
                drop table DELTA_p;
                drop table delta_p;
        }
}
```

CHAPTER 10

Demonstration Plan

This chapter describes the VLPDF demonstration plan. The demonstration will consist of three experiments designed to demonstrate the motivation and functionality of a D/KBMS and the components of a D/KBMS architecture. The next chapter describes several tests designed to study the relative tradeoffs in D/KBMS design and the factors contributing to D/KB query compilation, execution, and update times.

The chapter is organized as follows. Sections 10.1 and 10.2 describe the demonstration data base and rule base respectively. Sections 10.3 through 10.5 describe the demonstration experiments. For each experiment, we describe the objective, the background, and the experimentation procedure.

10.1. Demonstration Data Base

The demonstration database will consist of the following relations:

parent(childname char(20), parentname char(20)).

person(name char(20), sex char(20)).

parts(partid integer, name char(20), units char(10), weight integer).

supplier(name char(20), address char(50), supplierid integer).

b1(col1 integer, col2 integer).

b2(col1 integer, col2 integer).

b3(col1 integer, col2 integer).

b4(col1 integer, col2 integer).

b5(col1 integer, col2 integer).

10.2. Demonstration Rule Base

The demonstration rule base will consist of 1000 rules, 11 of which are listed below, the rest being random generated.

Ancestor rules

R_1: $anc(X, Y) - parent(X, Y)$.

R_2: $anc(X, Y) - parent(X, Z), anc(Z, Y)$.

Non-linear version of ancestor rules

R_3: $anc\_nl(X, Y) - parent(X, Y)$.

$$R_4: \quad anc\_nl(X, Y) - anc\_nl(X, Z), anc\_nl(Z, Y).$$

Rules with multiple cliques and mutual recursion

$$R_5: \quad p(X, Y) - p_1(X, Z), q(Z, Y).$$

$$R_6: \quad p(X, Y) - b_3(X, Y).$$

$$R_7: \quad p_1(X, Y) - b_1(X, Z), p_1(Z, Y).$$

$$R_8: \quad p_1(X, Y) - b_4(X, Y).$$

$$R_9: \quad p_2(X, Y) - b_2(X, Z), p_2(Z, Y).$$

$$R_{10}: \quad p_2(X, Y) - b_5(X, Y).$$

$$R_{11}: \quad q(X, Y) - p(X, Z), p_2(Z, Y).$$

10.3. Experiment 1: D/KBMS Motivation and Basic Functionality

Objective

- Demonstrate the motivation for combining Knowledge Based Systems (KBSs) and DBMS technologies to yield a D/KBMS — efficiently managing access to large, shared data/knowledge bases and improving productivity and functionality of information systems.

- Demonstrate the basic functions of a D/KBMS — knowledge representation, D/KB query language, D/KB query processing, and D/KB updates.

Background

Both KBS and DBMS technologies are designed for access and manipulation of information. Each has the potential for increasing the productivity and ease of use of the other. KBS technology provides techniques for acquiring and representing domain knowledge. It provides increased functionality for information systems via knowledge directed reasoning. It provides increased productivity for information systems via its clean separation between the knowledge base and inference engine. This separation allows incremental incorporation of new knowledge as it becomes available without changing the reasoning mechanisms. It also allows certain information retrieval tasks requiring application program development using a DBMS to be expressed as D/KB queries. This advantage becomes very apparent in information retrieval tasks involving recursive queries. These tasks require application program development when using a relational DBMS, since relational algebra is incapable of expressing recursive queries. However, using a data/knowledge base management system they can be expressed

simply as queries, since such a system can process recursive requests. The added functionality of a D/KBMS results in productivity gains for the end user.

Thus far, KBS technology has not addressed systems and efficiency issues. The knowledge bases in current KBS applications are small (typically, memory-resident) and difficult to share between applications. DBMS technology, on the other hand, offers solutions to various systems issues such as concurrency, security, integrity, reliability, and recovery. It offers solutions to various efficiency issues such as organizing large amounts of data and search and query optimizations.

Thus, KBS and DBMS technologies have much to offer each other — DBMS technology solutions to systems and efficiency issues and KBS technology improved functionality and productivity. This synergy is the motivation for combining these technologies and has led to the notion of a D/KBMS, a tool for efficiently managing access to large, shared data/knowledge bases.

Procedure

1. Look at the contents of the shared data/knowledge base, noting its size.

2. Demonstrate the database representation and manipulation facilities of a D/KBMS. The VLPDF demonstration testbed offers logic as the knowledge representation — specifically, Horn clauses with no complex terms and no negation. This representation provides the power of expressing relational operators in way simpler than the *de facto* relational data manipulation language, SQL.

   2a. *Demonstrate a selection operation* — "find all female persons" — by entering and executing the query:

   $$query(X) - person(X, "female").$$

   2b. Demonstrate a projection operation — "find the name and weight of every part" — by entering and executing the query:

   $$query(X, Y) - parts(NA1, X, NA2, Y).$$

   2c. Demonstrate a join operation — "find part name and supplier of every part" — by entering and executing the query:

   $$query(X, Y) - parts(Z, X, NA1, NA2), supplier(Y, NA3, Z).$$

3. Demonstrate the knowledge representation and modeling facilitites of a D/KBMS. Look at some of the rules in the data/knowledge base to get an idea of the type of knowledge that can be represented.

   3a. Demonstrate the D/KB query language by entering and executing the query:

   $$query(X) - anc("chris", X).$$

   This query means "find all ancestors of Chris".

   3b. Enter and execute the query:

   $$query(X) - anc(X, "chris").$$

This query means "find all persons whose ancestor is Chris". Comparing this query with the previous one demonstrates the expressive power of the Horn clause based query language.

4. List the rules in the workspace to see the definition of the *anc* relation.

5. Look at the application program needed to determine all ancestors of a given person when using a relational DBMS. Compare the size of this program with the two rules defining the *anc* relation to get an idea of the productivity improvement made possible by a D/KBMS.

6. Demonstrate the ease with which new knowledge can be added by entering the non-linear version of the ancestor rules (rules R_3 and R_4) into the workspace D/KB.

7. Enter and execute the query:

$$query(X) - anc\_nl("debby", X).$$

8. List the workspace D/KB rules.

9. Update the stored D/KB.

10.4. Experiment 2: D/KBMS Query and Update Processing Scenario

Objective

Demonstrate the various steps involved in processing queries and updates to a large, shared data/knowledge base.

Background

See chapter 7.

Procedure

1. Enter the following rule into the workspace D/KB:

$$femanc(X, Y) - anc(X, Y), person(Y, "female").$$

2. Enter the following query:

$$query(X) - femanc("eugene", X).$$

3. Demonstrate each step for compiling queries by describing its purpose and the changes it causes in the Knowledge Manager's data structures — rules, predicates, and query execution environment.

4. Execute the above query. Demonstrate the execution of the C code fragment by describing the information loaded into the object program's data structures. Demonstrate the execution of the LFP evaluation routine by describing the results of each step of the semi-naive evaluation algorithm.

5. Enter rules R_9 through R_{15} into the workspace D/KB and then update the stored D/KB. Demonstrate each step in update processing by describing its purpose and the changes it causes in the Knowledge Manager's data structures. Also, look at the contents of the rule storage structures, i.e., the relations *irulesource*, *ireachablepreds*, *isystables*, and *isyscolumns*.

10.5. Experiment 3: D/KB Query Language Embedded in PARLOG

Objective

Demonstrate the embedding of Horn clauses in PARLOG.

Background

PARLOG is a language with two types of relations: single-solution relations and all-solutions relations. All-solutions relations can appear in the body of single-solution relations. The evaluation semantics for these relations are completely different. Indeed, they can be viewed as two different languages — all-solutions relations constituting a query language and single-solution relations a parallel applications programming language. An all-solutions relation query can be evaluated by computing the least fixed point of the Horn clauses defining this relation. Thus, PARLOG can be considered as having a Horn clause query language embedded within it. This experiment will demonstrate this embedding.

Testbed Configuration

Same as for experiment 1.

Procedure

First, load the following single-solution program into the PARLOG environment.

```
mode partition(pivot?, list?, less-list`, greater-list`).

partition(u, [v|x1], [v|y1], z) <- v < u : partition(u, x1, y1, z).
partition(u, [v|x1], y, [v|z1]) <- u =<v : partition(u, x1, y, z1).
partition(u, [], [], []).


mode append(list1?, list2?, appended-list`), sort(list?, sorted-list`).


append([hd|tail], list2, [hd|atail]) <- append(tail, list2, atail).
append([], list2, list2).
```

```
sort([hd|tail], sorted-list) <-
            partition(hd, tail, list1, list2),
            sort(list1, sorted-list1),
            sort(list2, sorted-list2),
            append(sorted-list1, [hd|sorted-list2], sorted-list).
sort([], []).
```

Second, enter the following single-solution PARLOG query.

(z <= Allsol(Anc("eugene", x), Person(x, "female"))) & SET(y, x, z), Sort(y, a) &

This PARLOG query will produce a sorted list of the female ancestors of *eugene*. Here, the D/KB query *Anc("eugene", x), Person(x, "female")* is embedded in the single-solution PARLOG relation *SET*. The D/KB query returns a list of the female ancestors and binds *y* to this list. The *sort* call is like executing an application program.

Execute the above PARLOG query, demonstrating the various parallel processes created and the interactions between them.

CHAPTER 11

D/KBMS Performance Measurement and Evaluation

The demonstration experiments described in the previous chapter demonstrated the motivation and functionality of a D/KBMS and the components of a D/KBMS architecture. This chapter describes several experiments we designed to quantitatively measure D/KBMS performance and to understand D/KBMS performance sensitivity and behavior with respect to various system parameters. The basic motivation for doing these experiments is to justify the D/KBMS architecture specification methodology described in chapter 8. That is, to show that this methodology can indeed be used to design high performance D/KBMSs.

The chapter is organized as follows. Section 11.1 describes D/KBMS performance measures. Section 11.2 describes parameters that affect these measures. Sections 11.3 and 11.4 characterize the data and rule bases used during the experimentation. Section 11.5 contains a description of the tests and an analysis of the test results. Section 11.6 describes the conclusions drawn from the experimentation.

11.1. D/KBMS Performance Measures

The main D/KBMS performance measures are shown in table 11.1.

t_c	D/KB query compilation time
t_e	D/KB query execution time
t_u	D/KB update time

Table 11.1. D/KBMS performance measures

During experimentation, values of these measures were obtained by averaging over 5 readings.

The components comprising D/KB query compilation time, t_c, are shown in table 11.2. Each component corresponds to a step in the compilation procedure. We haven't included the time to execute the magic set optimization algorithm, since we have not implemented this algorithm in our testbed.

t_{c1}	Time to parse the query.
t_{c2}	Time to construct the query execution environment data structure.
t_{c3}	Time to extract the relevant predicates from the workspace D/KB.
t_{c4}	Time to extract the relevant rules from the stored D/KB.
t_{c5}	Time to extract the relevant predicates from the workspace D/KB after the relevant rules from the stored D/KB have been extracted.
t_{c6}	Time to read the D/KB data dictionaries.
t_{c7}	Time to find the cliques in the workspace D/KB rules.
t_{c8}	Time to generate the evaluation order list, i.e., to construct the evaluation order graph and perform a topological sort of this graph.
t_{c9}	Time to perform the semantic checks.
t_{c10}	Time to generate the code fragment.
t_{c11}	Time to compile the code fragment and link it with the run-time library.

Table 11.2. t_c breakup

The components comprising D/KB query execution time are shown in table 11.3. Each component corresponds to a step in LFP evaluation.

t_{e1}	Time for initialization.
t_{e2}	Time for while loop execution.
t_{e3}	Time for dropping derived predicate and other temporary tables.

Table 11.3. t_e breakup

The components comprising D/KB update time are shown in table 11.4.

t_{u1}	Time to extract the rules relevant to the Workspace D/KB from the Stored D/KB.
t_{u2}	Time to update the *irulesource* relation.
t_{u3}	Time to update the *ireachablepreds*, *isystables*, and *isyscolumns* relations.

Table 11.4. t_u breakup

11.2. Parameters

We studied the effect of several parameters on the three D/KBMS performance measures described in the previous section. These parameters can be grouped into three categories: (i) D/KBMS architecture related parameters, (ii) workload related parameters, and (iii) D/KB query and update related parameters. The D/KBMS architectural parameters relate to different aspects of the system architecture, the workload parameters relate to D/KB size, and the query and update parameters relate to the portions of the D/KB relevant to the query. Table 11.5 describes these parameters.

11.3. Data Base Characterization

The base relations used in the experimentation are all binary relations. We characterize them in terms of their directed graph representation. In this representation, a binary relation is represented as a directed graph; domain elements form the nodes of this graph and tuples the edges.

We used the following types of data bases in the experimentation: lists, full binary trees, directed acyclic graphs, and directed cyclic graphs. The parameters used to characterize these data bases are shown in table 11.6.

Relation type	Parameters	Remarks
List	Number of lists, their average length	The number of tuples in a data base with n lists of average length l is approximately $n(l - 1)$.
Full binary tree	Number of trees, their depth	The number of tuples in a data base with n trees, each of depth d, is $n(2^d - 2)$.
Directed acyclic graph	Number of tuples, fan-out, fan-in, path length	Fan-out and fan-in respectively refer to the number of arcs leaving and entering a node in the graph. Path length refers to the number of nodes in a path starting with a node with zero fan-in and ending with a node with zero fan-out.
Directed cyclic graph	Number of tuples, fan-out, fan-in, path length, number of cycles in the graph, their average length	

Table 11.6. Database characterization

To facilitate experimentation, we developed a D/KB generator that accepts parameter values as input and creates a "random" data base satisfying these values.

11.4. Rule Base Characterization

We characterize the rule base by characterizing its PCG, which is a directed cyclic graph. Table 11.7 shows the parameters we used to characterize the PCG. The D/KB generator accepts these parameters as input and creates a "random" directed cyclic graph satisfying these parameters. It then generates a rule base, such that the PCG of

	Parameter	Description
D/KBMS architecture related parameters		Optimization strategy
		LFP evaluation strategy
		Rule storage structures
Workload related parameters	R_s	Total number of rules in the Stored D/KB
	R_w	Total number of rules in the Workspace D/KB
	P_s	Total number of derived predicates in the Stored D/KB
Query and update related parameters	R_{sr}	Number of Stored D/KB rules relevant to the query
	P_r	Number of derived predicates relevant to the query
	D_{sr1}	Total number of tuples in all base relations relevant to the query
	D_{sr2}	Number of relevant base relation tuples
	T_w	Number of edges in the transitive closure of the PCG of the Workspace D/KB rules after extracting the relevant rules from the Stored D/KB

Table 11.5. Parameters

Number of cliques

Average number of derived predicates in each clique

Average number of rules defining a derived predicate

Probability that a rule calls a base relation

Fan-out

Fan-in

Path length

Number of cycles in each clique

Average cycle length

Table 11.7. Rule base characterization

this rule base is the above cyclic graph. In general, there are several rule bases satisfying this condition; the D/KB generator generates one such rule base.

11.5. Tests and Results

We now describe the results of the performance measurement and evaluation tests we performed using the testbed. These tests can be categorized into two groups: (1) tests relating to D/KB query processing and (2) tests relating to D/KB updates.

11.5.1. Tests Relating to D/KB Query Processing

The tests relating to D/KB query processing can be further categorized into two groups: (a) tests relating to D/KB query compilation and (b) tests relating to D/KB query execution.

11.5.1.1. Tests Relating to D/KB Query Compilation

In this section, we describe the tests we conducted relating to D/KB query compilation. After making several measurements of the various components contributing to the compilation time, we found that the parameters that had the most effect on D/KB query compilation time were R_s, P_s, R_{sr}, and P_r. R_s and R_{sr} affect the time to extract the relevant rules from the Stored D/KB, while P_s and P_r affect the time to read the D/KB data dictionaries. The purpose of the tests below is to study the effect of these

parameters on D/KB query compilation time.

(1) *Study the effect of the total number of rules in the Stored D/KB, R_s, and the number of Stored D/KB rules relevant to a query, R_{sr}, on the time to extract the relevant rules from the Stored D/KB, t_{c4}. We varied R_s from 29 to 205 in steps of 16, recording the value of t_{c4} for a query with $R_{sr} = 2$. We then repeated this procedure for queries with $R_{sr} = 7$ and $R_{sr} = 20$.*

Figure 11.1 shows the results of this experiment. For each query, notice that t_{c4} is relatively insensitive to R_s. To understand why this is so, let us look at how the relevant rules are extracted from the Stored D/KB for a typical D/KB query, say,

$$query(X, Y) - p(X, Z), m(Z, Y).$$

This is accomplished via the following SQL query:

SELECT irulesource.rule

FROM irulesource, ireachablepreds

WHERE (ireachablepreds.topredname = irulesource.headpredname OR

        ireachablepreds.frompredname = irulesource.headpredname) AND

        ireachablepreds.frompredname = "p" OR

        ireachablepreds.frompredname = "m"))

The insensitivity of t_{c4} to R_s (the number of tuples in *irulesource*) is because *irulesource* is typically a small enough relation to hold in memory and because *ireachablepreds* has an index on its columns.

Notice in figure 11.1 that for a given value of R_s, t_{c4} increases with R_{sr}, the number of rules in the Stored D/KB relevant to the query. This is because R_{sr} is related to the join selectivity of the above SQL query. Figure 11.2 shows a plot of t_{c4} versus R_{sr} for three different values of R_s.

The data in figure 11.1 is for the case where the rules are stored in compiled form in the Stored D/KB as the transitive closure of the PCG. If they are stored in raw source form only, or if the compiled form is just the PCG (as opposed to its transitive closure), the transitive closure of the PCG would have to be computed during query compilation. We did not make quantitative measurements of t_{c4} versus R_s for these cases, since we know from our previous work on algorithms for computing the transitive closure of a database relation that the performance deteriorates rapidly for very large relation sizes (see chapter 7).

(2) *Study the effect of the total number of derived predicates in the Stored D/KB, P_s, and the number of derived predicates relevant to the query, P_r, on the time to read the D/KB data dictionaries.* The purpose of reading these dictionaries is to determine the types of the columns of the base and derived predicates prior to executing the type inferencing algorithm. The motivation for doing this experiment is that reading the D/KB data dictionaries involves accessing the Stored D/KB, which impacts performance. The procedure here was basically the same as that in the preceding experiment. The values of P_r were 1, 4, and 10 for the three queries.

Figure 11.3 shows a plot of t_{c6} versus P_s. Notice that for a given value of P_r, t_{c6} is relatively insensitive to P_s. To see why this is so, let us look at how the intensional data dictionary is read for a query having say, p_1 and p_2 as the relevant derived predicates. This is accomplished via the following SQL query:

SELECT *

FROM isystables, isyscolumns

WHERE isystables.tabid = isyscolumns.tabid AND

        (isystables.tabname = "p_1" OR

        isystables.tabname = "p_2"))

The execution time of the above query is insensitive to P_s (the number of tuples in *isystables*), because we place indexes on *isystables* and *isyscolumns*.

Also notice that for a given value of P_s, t_{c6} increases with P_r. This is because P_r is related to the join selectivity of the above query. Figure 11.4 shows a plot of t_{c6} versus P_r for three different values of P_s.

We haven't shown a plot of t_{c6} versus R_s, but we found t_{c6} to be insensitive to R_s. This is because R_s affects t_{c6} in so far as it affects P_s, the number of derived predicates in the Stored D/KB, and as we explained above, t_{c6} is insensitive to P_s.

(3) *Study the relative contributions of the different steps in D/KB query compilation on the total compilation time t_c.* After making several measurements of t_{c1} through t_{c11}, we found that the principal contributions to t_c came from t_{c2}, t_{c4}, t_{c6}, t_{c8}, and t_{c11}. Figure 11.5 shows pie charts of the contributions of these components for three different queries, with R_{sr} equal to 1, 7, and 20. Notice that as R_{sr} increases from 1 to 20, the relative contribution of t_{c4}, the time to extract the relevant Stored D/KB rules, increases from 25% to 67%. Also, the rate of increase appears to be quite rapid.

t_{c8} appears to be making a non-trivial contribution to t_c. This is actually due to

the fact that in our testbed, the evaluation order list is computed by making a Unix system call to execute the Unix topological sort utility. The overhead imposed by the system call is particularly significant, since the evaluation order graph is typically quite small. We could have avoided making the system call by implementing a topological sort algorithm; this would have had the effect of making the contribution of t_{c8} insignificant.

Finally, note that the relative contribution of t_{c11}, the time to compile the code fragment generated by the Knowledge Manager and link it with the run time library appears quite significant. However, this is very much compiler dependent and can vary greatly from system to system. We can make a similar observation about t_{c2}.

11.5.1.2. Tests Relating to D/KB Query Execution

Quantitative analysis of D/KB query execution performance is complicated by the fact that the execution time is greatly influenced by the nature of the query and data. This is because the type of query and data greatly affect the size of the set of relevant facts, D_{sr2}, and the amount of duplicate work done during LFP computation, which were shown in [Banc86] to be two of the most important parameters influencing D/KB query execution time. The principal purpose of the tests described in this section is to study the effect of these parameters on D/KB query execution time.

The tests all use the *ancestor* query:

$ancestor(X, Y) - parent(X, Y).$

$ancestor(X, Y) - parent(X, Z), ancestor(Z, Y).$

$query(X) - ancestor("john", X).$

and tree structured data for the *parent* relation. The results will obviously be different for other queries and data types. Still, we can draw some general conclusions from the test results for the *ancestor* query and tree structured data.

When studying the effect of redundant work, we didn't directly measure this parameter. Rather, we measured the execution times for naive and semi-naive LFP evaluation, the difference between the two indicating the impact of redundant work.

We now describe the tests in more detail.

(4) *Study the effect of the fraction of relevant facts, D_{sr2}/D_{sr1}, on D/KB query execution time, t_e.* We varied D_{sr2}/D_{sr1} in two different ways. In the first method, we kept D_{sr1} fixed by keeping the size of the *parent* relation fixed and varied D_{sr2} by rooting the *ancestor* query at different sub-trees of the *parent* relation. Thus, each value of D_{sr2} was obtained from a different query, each query having a different constant. In the second method, we kept D_{sr2} fixed by fixing the query constant and varied the size of D_{sr1} by making the *parent* relation progressively larger.

Here, the same query was applied to *parent* relations of different sizes. Semi-naive evaluation was used for LFP computation. Optimization was not used.

Figure 11.6 shows a plot of t_e versus D_{sr2}/D_{sr1}. Notice that when D_{sr1} is fixed, t_e is insensitive to D_{sr2}. This is because in the absence of the magic set optimization, the transitive closure of the entire *parent* relation is computed, regardless of the percentage of this relation that is actually relevant to the query. We study the impact of this optimization in another experiment.

On the other hand, when D_{sr2} is fixed but D_{sr1} is not, t_e increases with D_{sr1} (equivalently, t_e decreases with D_{sr2}/D_{sr1}). This is because the transitive closure is being computed for progressively larger relation sizes.

(5) *Study the impact of redundant work done during LFP computation on D/KB query execution time.* We first measured t_e for several *ancestor* queries rooted at different sub-trees of the *parent* relation, keeping D_{sr1} fixed and using semi-naive LFP evaluation. We then repeated this procedure for naive LFP evaluation.

Figure 11.7 shows a plot of t_e versus D_{sr2}/D_{sr1} for both naive and semi-naive evaluation. Notice that for the query and database in this test, semi-naive evaluation is between 2.5 to 3 times faster than naive evaluation. The difference is due to the fact that semi-naive evaluation avoids a lot of duplicate work by computing only the differential of $f(R)$ during each iteration when evaluating the LFP of $R = f(R)$. Naive evaluation, on the other hand, recomputes tuples computed in previous iterations.

(6) *Study the relative contributions of the various steps in the while loop of naive and semi-naive LFP evaluation.* Chapter 12 showed the pseudo-code for naive and semi-naive LFP evaluation used in our testbed. This experiment studies the relative contributions of the various steps in the while loop of these algorithms. Table 11.8 shows these steps.

Figure 11.8 shows the results of this test. Notice that in naive evaluation, 94% of the time is spent in evaluating the right hand side of the recursive equations and doing the termination check, while the corresponding activities consume 82% of the time in semi-naive evaluation. The activities are not quite the same since semi-naive evaluation computes only the differential of the right hand side of the recursive equations during each iteration.

Figure 11.9 shows a comparision of the time taken to evaluate the right hand side of the recursive equations (or the differential in semi-naive evaluation) and to do the termination check for naive and semi-naive evaluation. Notice that the times for naive evaluation are about 2.5 to 3 times greater than those for semi-naive evaluation. This is the principal reason semi-naive evaluation was found to be 2.5 to 3 times faster than naive evaluation in the previous test.

Step	Naive	Semi-Naive
t_{we0}	Time to make temporary copy	Time to clear temporary tables
t_{we1}	Time to evaluate right hand side of recursive equations	Time to evaluate differential of right hand side of recursive equations
t_{we2}	Time for termination check	Time to make temporary copy
t_{we3}	Time to clear temporary tables	Time to clear temporay tables
t_{we4}	Time to drop temporary tables and create new ones for next iteration	Time for termination check
t_{we5}		Time to make temporary copy

Table 11.8. Steps in while loop of LFP evaluation

The termination check is expensive in our implementation because the SQL interface between the Knowledge Manager and the DBMS forces a set difference, an expensive operation, to be computed during this check.

(7) *Study the impact of using the magic set optimization on D/KB query execution time.* This test consists of three parts. First, we measured t_e as a function of D_{sr2}/D_{sr1} for the four cases resulting from using naive and semi-naive evaluation with and without optimization. D_{sr2}/D_{sr1} was varied by keeping D_{sr1} fixed and varying D_{sr2}. Figure 11.10 shows the results of this test. Notice that t_e is insensitive to D_{sr2}/D_{sr1} in the absence of optimization, since the transitive closure of the entire *parent* relation is computed in this case. However, with optimization, t_e increases with D_{sr2}/D_{sr1}. This is because with optimization the transitive closure is computed only for the relevant portion of the *parent* relation, which grows progressively larger in this test.

The tradeoffs in using optimization can be clearly seen in figure 11.10: there is a crossover point beyond which optimization results in higher query execution times. This typically happens when the selectivity of the query is high, i.e., when most of the database is relevant. To understand why this is so, recall that in the magic set

strategy, an LFP computation is done is first to evaluate the magic rules and determine the set of relevant facts. Then, another LFP computation is done to evaluate the modified rules and determine the query results. In the latter computation, the magic set predicates are base relations. When the selectivity of the query is low, the size of these relations is small enough so that the two LFP computations together take less time than a single LFP computation on the original base relations. However, when the selectivity of the query is high, the size of the magic set predicates is large and the extra overhead in first computing them results in a higher overall execution time.

The crossover selectivity where optimization degrades performance for semi-naive evaluation is about 72%, while it is about 82% for naive evaluation. The higher crossover point for naive evaluation is due to the fact that optimization has a bigger impact on naive evaluation as it does a lot of redundant work.

Figure 11.11 shows the execution times for the two LFP computations as a function of D_{sr2}/D_{sr1}. The rate of increase of the magic rules evaluation is lower than that for the modified rules evaluation. This is because the magic rules evaluation time depends mostly on D_{sr1}, the size of the base relations, which was fixed in this test. On the other hand, the modified rules evaluation time is quite sensitive to D_{sr2}, the number of relevant facts, which was varied.

The impact of optimization is significant for queries with low selectivity. For example, notice from figure 11.10, that for semi-naive evaluation when only 5% of the base relation tuples are relevant, the execution time with optimization is about 6 to 7 times faster than without optimization.

The impact of optimization is particularly significant for very large base relations and very low query selectivity. The second part of this test studied this impact. Here, we executed an *ancestor* query with very low selectivity (.05%) against a *parent* relation containing over 16,000 tuples and measured t_e with and without optimization. We found that without optimization, the query took several orders of magnitude longer to execute than it did with optimization! We expect that in very large database environments, the query selectivity will be small in many cases and so, this test represents a very plausible scenario.

In the third part of this test, we varied D_{sr2}/D_{sr1} by keeping D_{sr2} fixed and varying D_{sr1}. Figure 11.12 compares the query execution times with and without optimization. As expected, with optimization the curve is flat, since the number of relevant facts is fixed. On the other hand, without optimization, the execution time decreases with D_{sr2}/D_{sr1}, since the transitive closure is being computed for progressively larger relation sizes.

11.5.2. Tests Relating to D/KB Updates

In this section, we describe the tests we conducted relating to D/KB updates. We reiterate that in our testbed, D/KB updates just update the rule storage structures in the Stored D/KB with the Workspace D/KB rules. In particular, there is no checking of these rules against integrity constraints that may be associated with the Stored D/KB.

The parameters affecting D/KB update time are R_s, R_{sr}, R_w, and T_w. R_s and R_{sr} affect the time taken to extract the relevant rules from the Stored D/KB, while R_w and T_w affect the time taken to update the *ireachablepreds* storage relation. The experiments below study the effect of these parameters.

(8) *Study the effect of R_s and R_{sr} on the D/KB update time, t_u.* We loaded the Workspace D/KB with a single rule and updated the Stored D/KB, varying the value of R_s from 9 to 189. There were 8 rules in the Stored D/KB relevant to the Workspace D/KB rule. Figure 11.13 shows a plot of t_u versus R_s both with and without compiled rule storage structures. In the latter case, the update time is simply t_{u3}, the time to store the source form of the rules. Notice that updates are almost an order of magnitude faster without compiled form rule storage.

Also, t_u is relatively insensitive to R_s. The main reason for this is that the time to extract the relevant rules from the Stored D/KB is a significant contributor to t_u (see next experiment) and this time depends only on R_{sr} (see D/KB query compilation experiment 1). We have not explicitly studied the impact of R_{sr} on t_u as it is the same as the impact of R_{sr} on t_{c4}, which we studied before. The insensitivity of t_u to R_s is significant because it means that the D/KB update time does not degrade for very large rule sets.

(9) *Study the relative contributions of the different components of t_u as a function of R_w and T_w.* We updated a Stored D/KB with $R_s = 189$ with a Workspace D/KB containing 38 rules and measured the values of t_{u1}, t_{u2}, and t_{u3}. The value of T_w was 137, i.e., the transitive closure of the PCG of the Workspace D/KB rules after extracting the relevant rules contained 135 edges. We then repeated this procedure for a Workspace D/KB with 1 rule and $T_w = 21$. Figure 11.14 shows a pie chart of the relative contributions. The main point to note is that t_{u1}, the time to extract the relevant rules is a significant component of t_u. For small values of R_w and T_w, this time in fact makes the bulk of the contribution to t_u. However, for large values of R_w and T_w, the percentage contribution of t_{u1} decreases, since that of t_{u3} increases. Also, notice that the time to store the source form of the rules, t_{u2}, contributes only a small amount to the overall D/KB update time.
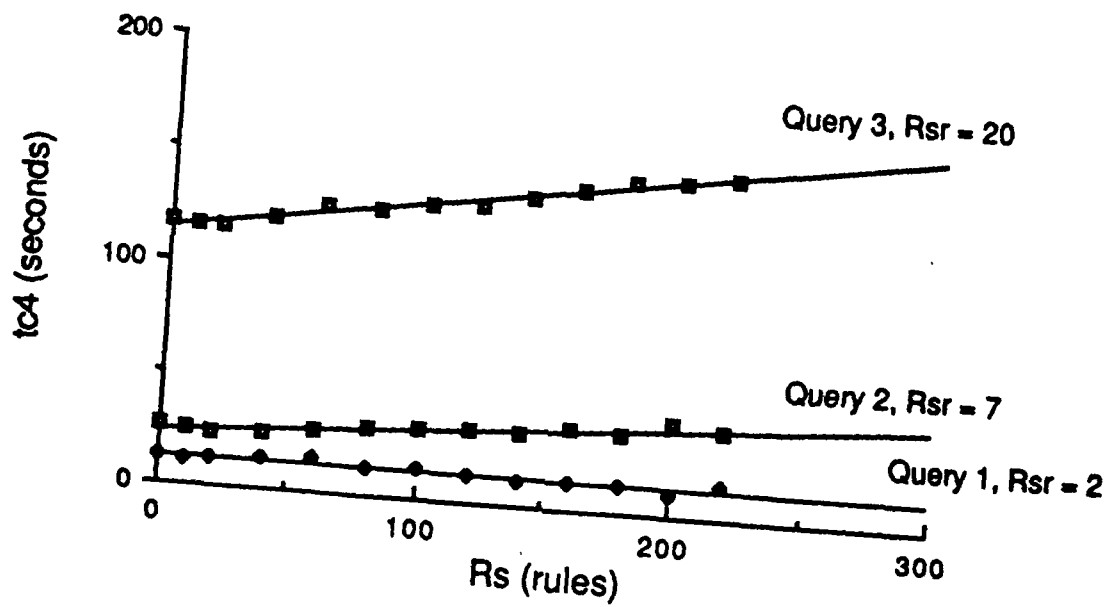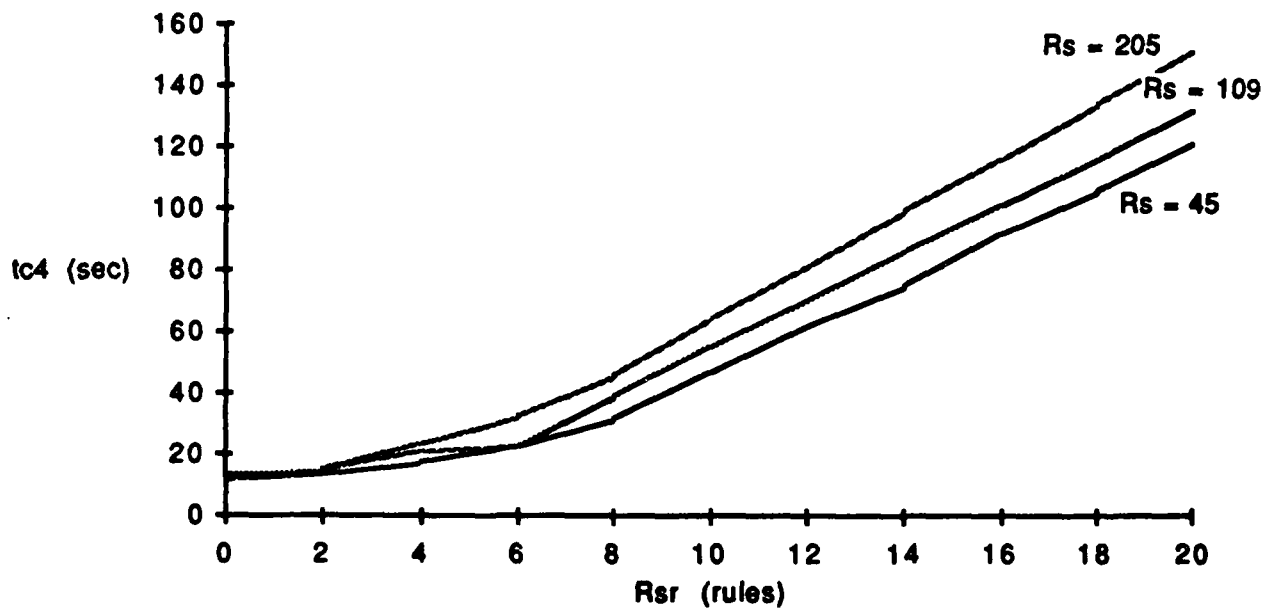
Figure 11.1. t_{c4} vesus R_s
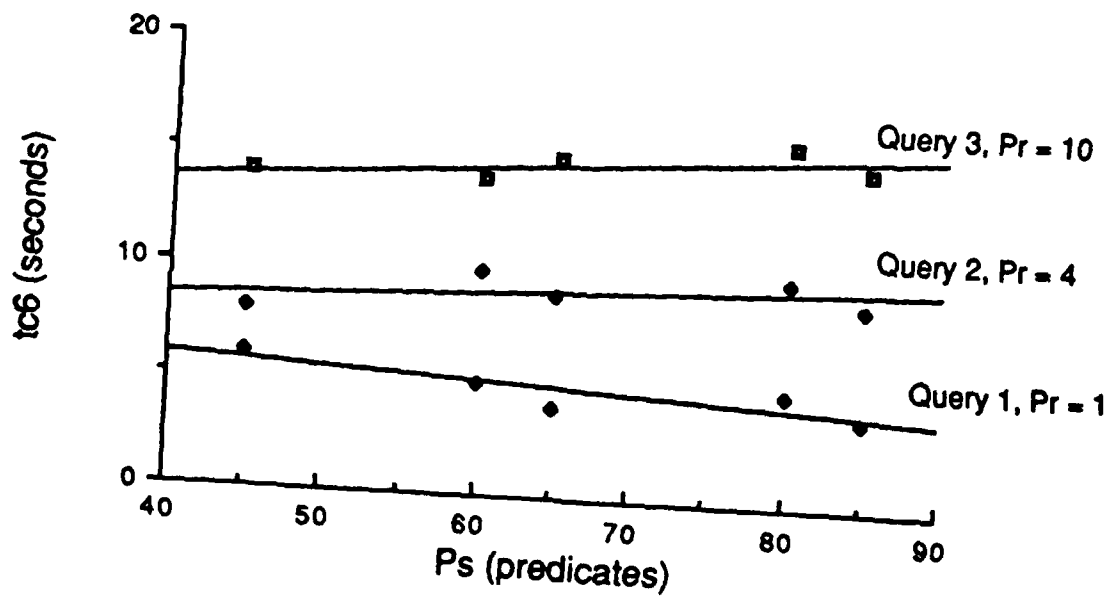
Figure 11.2. t_{c4} versus R_{sr}
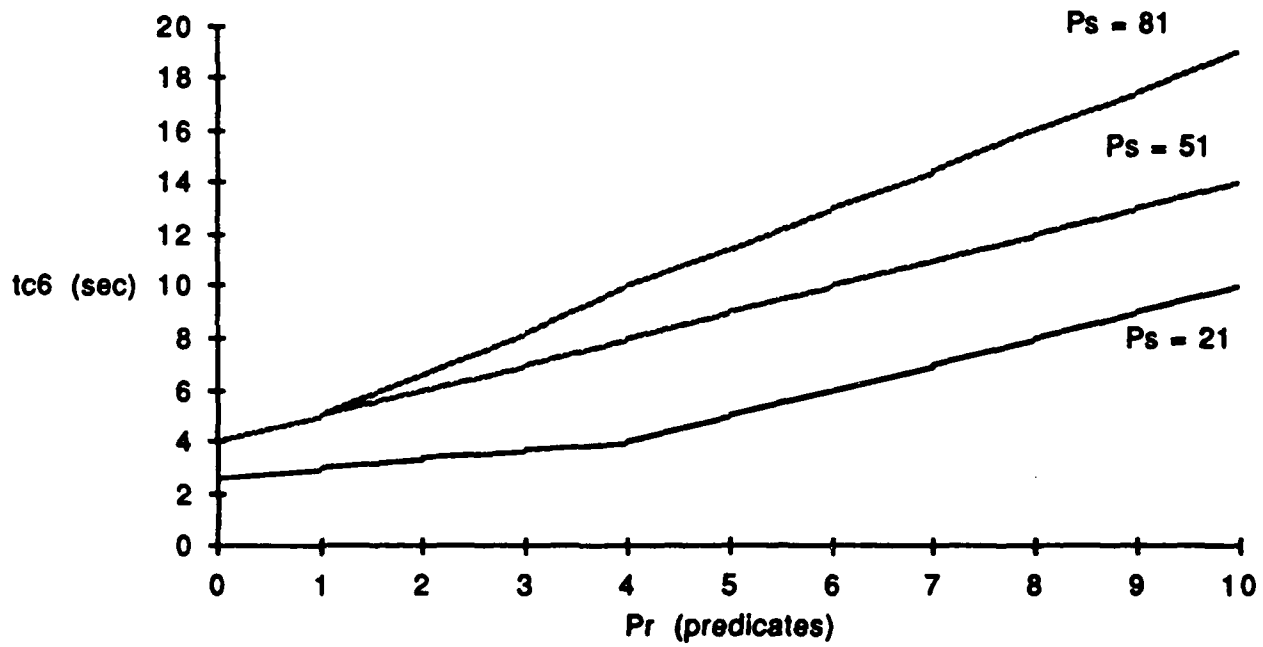
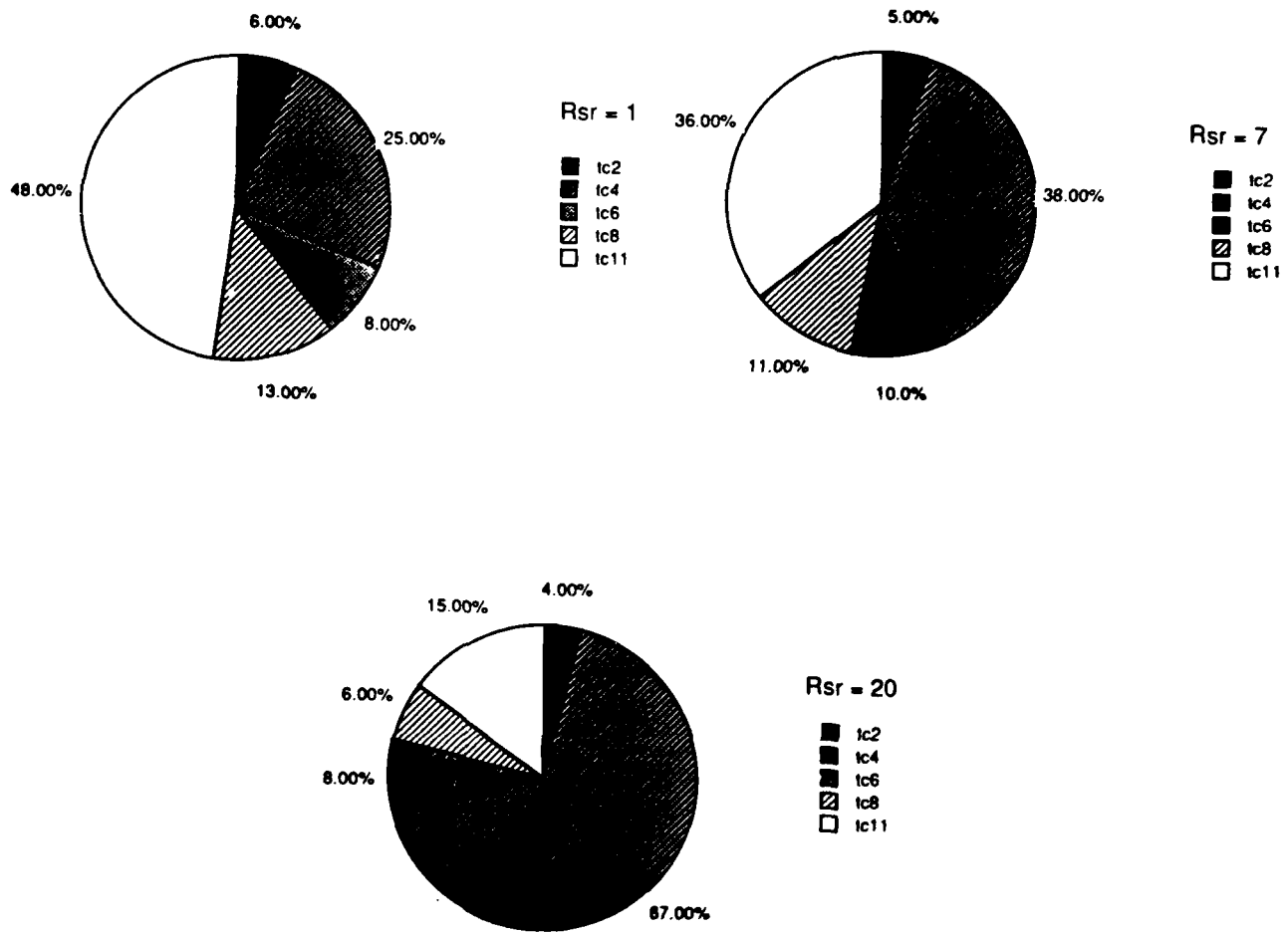Figure 11.3. t_{c6} versus P_s

Figure 11.4. t_{c6} versus P_r

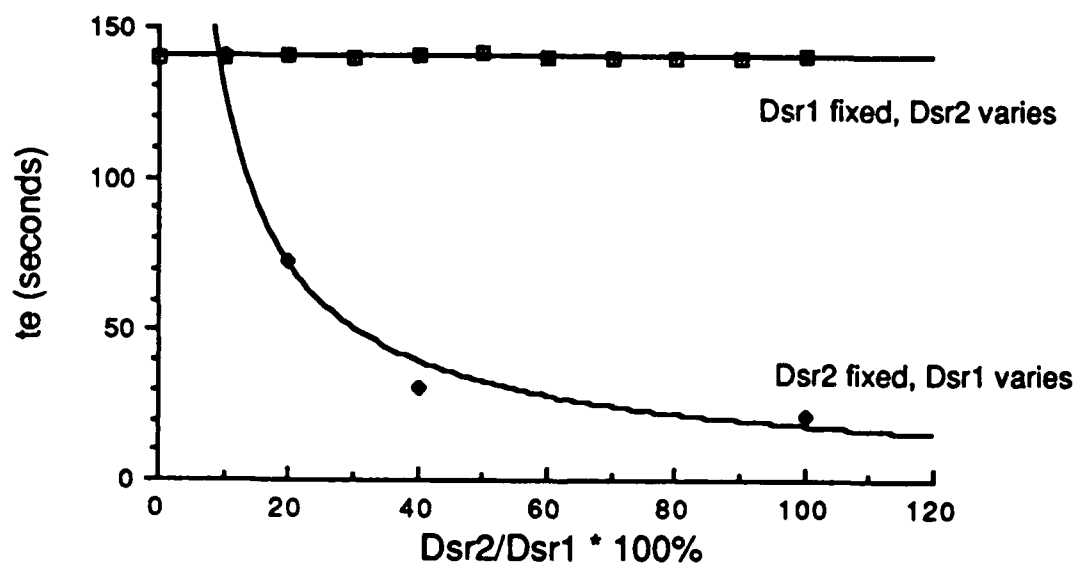Figure 11.5. Breakup of D/KB query compilation time

Figure 11.6. t_e versus D_{sr2}/D_{sr1}
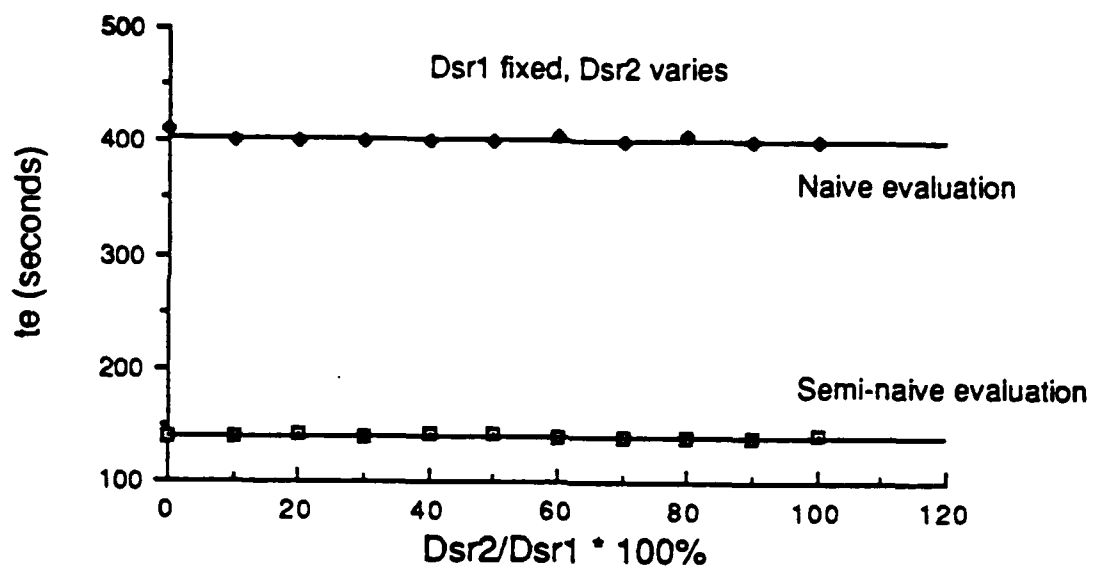
Figure 11.7. Comparison of naive and semi-naive LFP evaluation
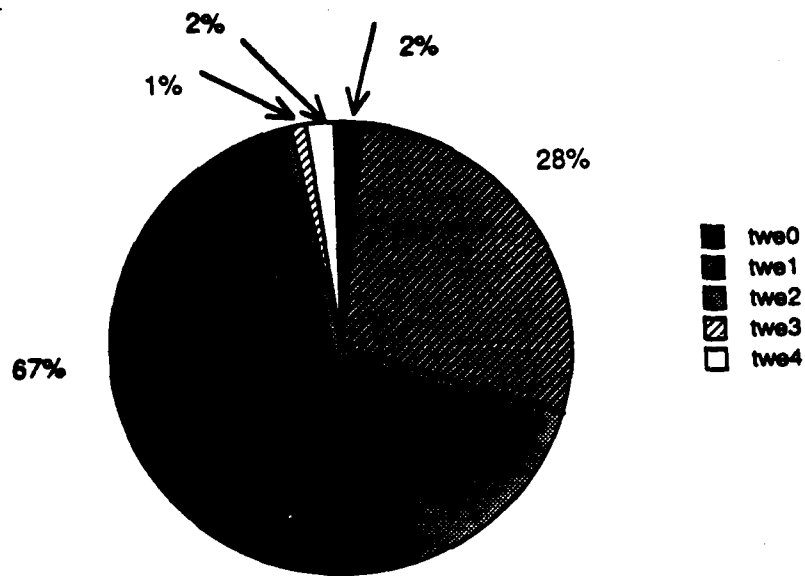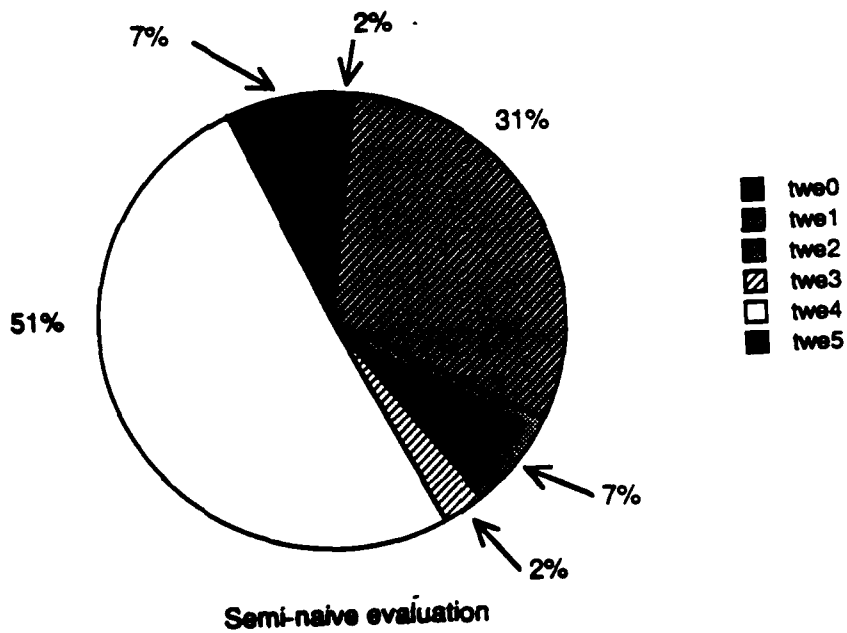
Figure 11.8. Breakup of LFP while loop evaluation

t1 = Time for f(R) or differential of f(R) evaluation

t2 = Time for termination check
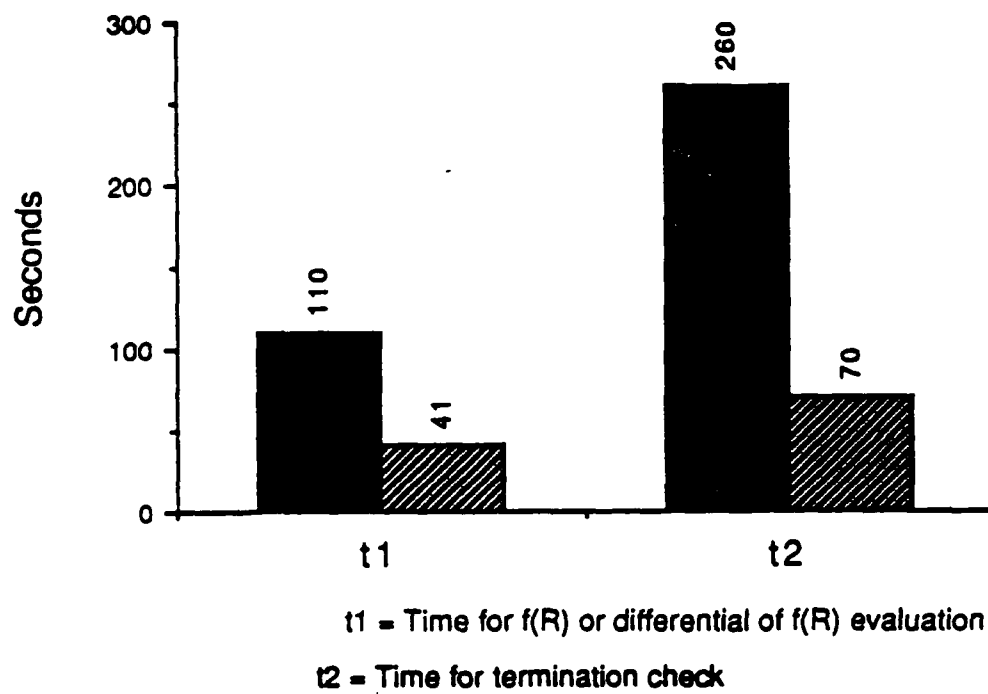
Figure 11.9. LFP termination check comparison

Figure 11.10. Effect of optimization for fixed D_{sr1}

Figure 11.11. Magic and modified rules evaluation time

Figure 11.12. Effect of optimization for fixed D_{sr2}

Figure 11.13. t_u versus R_s

Figure 11.14. Breakup of D/KB update time

11.6. Conclusions

This chapter presented and analyzed the results of several experiments we performed to quantitavely measure D/KBMS performance and to study performance as a function of various parameters. We list below the conclusions we can draw from the D/KBMS performance measurement and evaluation study.
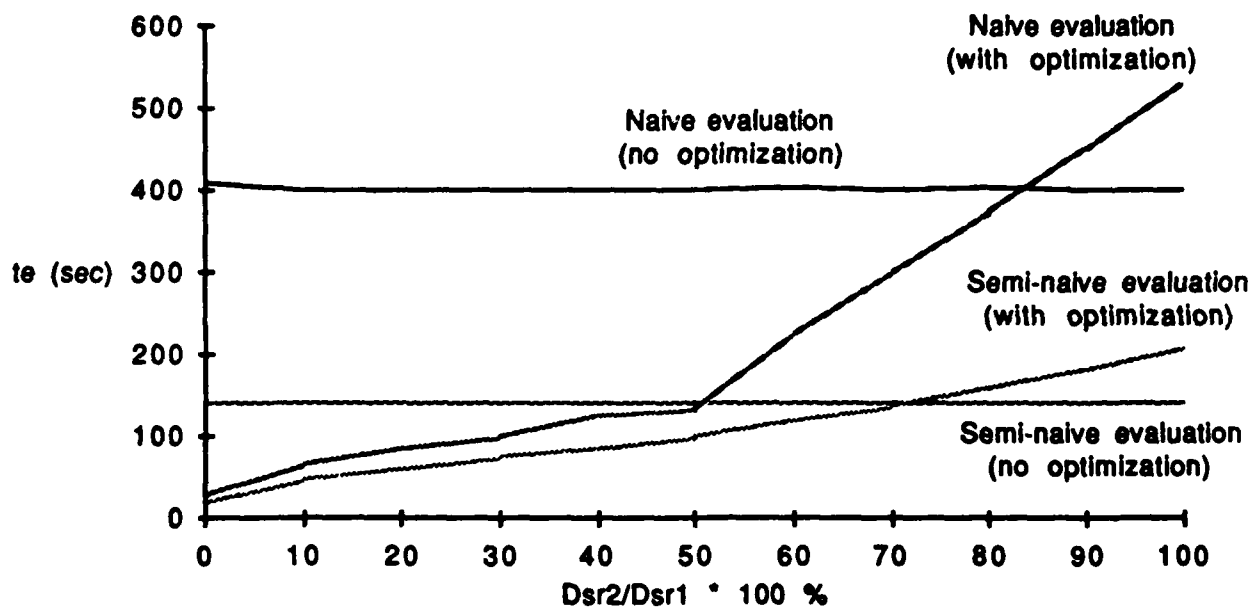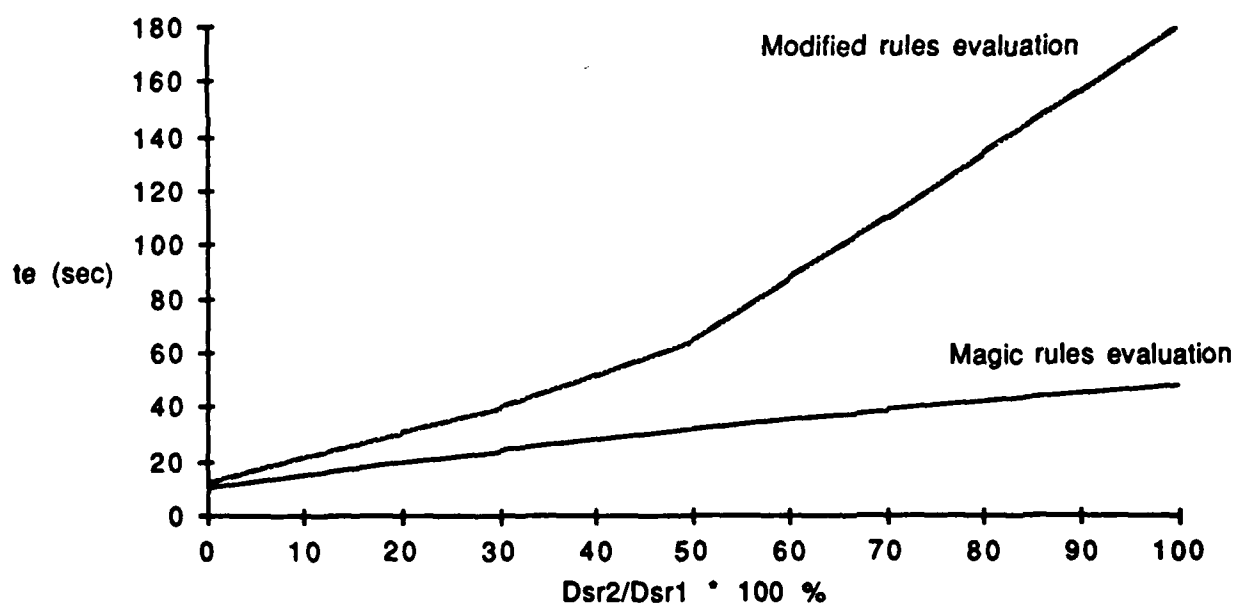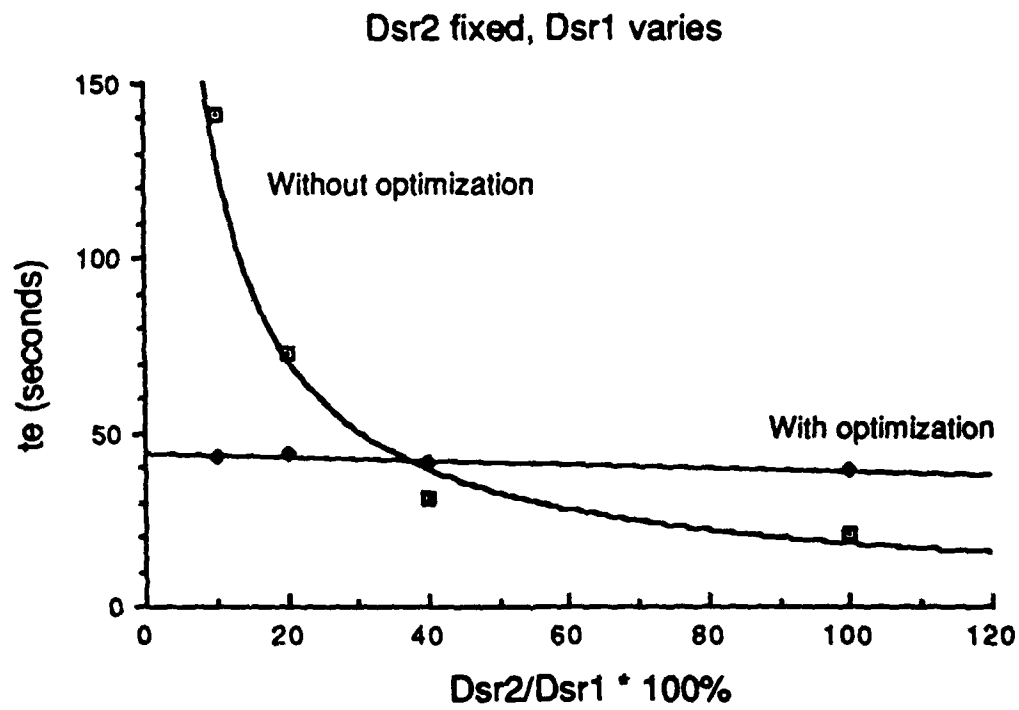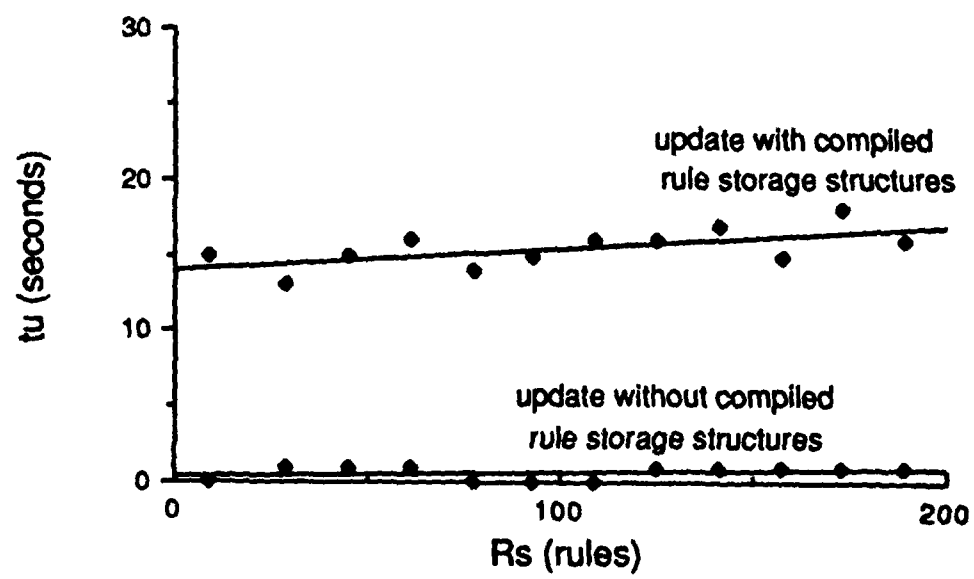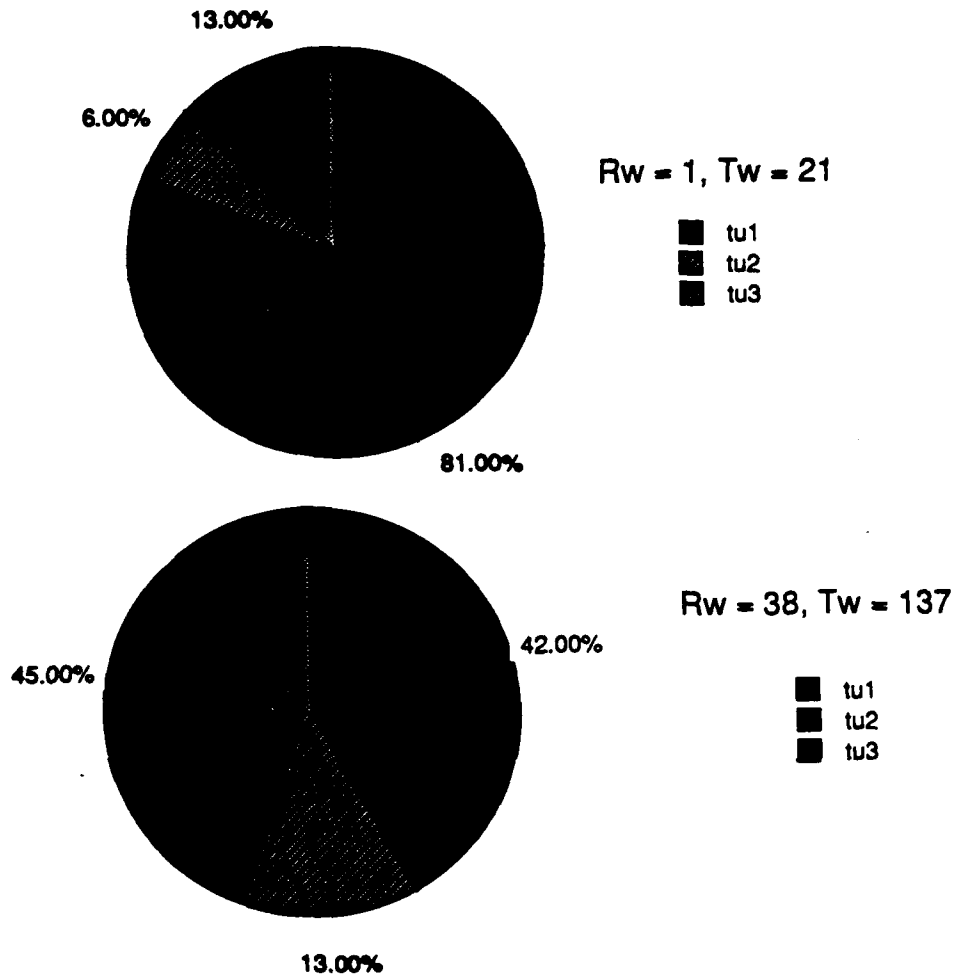
1. In order that the D/KBMS be scalable to handle large rule sets. it is important that the rule storage structures be such that the time to extract the relevant rules is independent of the total number of rules in the Stored D/KB. Otherwise. the D/KB query compilation times will grow with the size of the rule base. Our experimentation has shown that storing the transitive closure of the PCG of the rules and placing an index on the columns of this storage structure achieves the effect of making the relevant rules extraction time independent of the rule base size. This is because with this storage structure. the time to extract the relevant rules depends only on the number of rules extracted and not on the total number of rules.

2. There are two important tradeoffs that relate to rule storage structures. The first is a time-vs-space tradeoff. Compiled form rule storage structures like the transitive closure of the PCG use more space but permit faster query compilation than non-compiled storage structures. The other tradeoff is between query compilation time and update time. Compiled form storage structures take longer to update. sometimes even an order of magnitude longer as some of our experiments indicated. than non-compiled storage structures. The choice of rule storage structure must be dictated by the relative cost of storage versus compilation and by the application characteristics – whether it is query intensive or update intensive.

3. The PCG itself (as opposed to its transitive closure) has been proposed as a rule storage structure. However. this is not a good choice for query intensive applications. This is because during query compilation. the transitive closure of the PCG will have to be computed to extract the relevant rules and this can get very time consuming for rules with large PCGs.

4. As we argued before. from the D/KB query compilation point of view. a good rule storage structure is one where the relevant rule extraction time depends only on the number of rules extracted and not on the total number of rules. However. we found that the time to extract the relevant rules is *very* sensitive to the number of rules extracted. A key to avoiding excessive compilation times is to structure the rules in such a way that the number of relevant rules for a query is small. Object-oriented database techniques can prove to very useful here. For example. a small set of rules can be encapsulated within an object and these rules can be retrieved whenever the object receives a message representing a query against them. Encapsulating rules within an object is a way of structuring the rules so that only the relevant portions of the rule base are processed during compilation. Of course. much work needs to be done to integrate the concepts of object-oriented database systems – inheritance, message processing, persistent objects, etc. – with those of D/KB query and update processing.

5. Precompilation of D/KB queries can prove to be very useful. This is especially true for frequently occurring queries with large R_{sr} values. The price of precompilation is that, for precompiled queries, information about referenced relations and rules must be recorded. During updates, this information is checked to see whether the update invalidates any compiled query. However, for applications involving few updates and frequently occurring queries with large R_{sr} values, this price is well worth paying.

6. Two of the main parameters affecting D/KB query execution time are the ratio of relevant facts to total number of facts (D_{sr2}/D_{sr2}) and the amount of redundant work done in the while loop of LFP evaluation. To reduce the amount of redundant work and to restrict the LFP evaluation to the relevant database tuples, the D/KBMS architecture can use semi-naive LFP evaluation and the generalized magic sets optimization strategy.

7. There is a tradeoff in using optimization: while optimization restricts LFP evaluation to the relevant tuples of the database, work must be done to first determine these tuples. There is a crossover value of D_{sr2}/D_{sr1} beyond which optimization actually results in higher query execution times. Optimization pays best when the selectivity of the query is low, i.e., for queries that retrieve only a small fraction of the database. The benefit of optimization is particularly telling for queries with very low selectivity and very large base relations. For such applications, we found that without optimization, the query took several orders of magnitude longer to execute than it did with optimization! We expect that in very large database environments, the query selectivity will be small in many cases and the use of optimization is highly recommended despite the extra work introduced. Ideally, the D/KBMS query optimizer should adapt the optimization strategy dynamically, switching it on for queries with low selectivity and off otherwise.

8. Relational algebra alone is not a good choice for the DBMS interface, since the LFP evaluation in this case has to be done via an application program and this introduces several inefficiencies. For example, during each iteration of the while loop, several table copies are performed. Also, the termination check becomes a very expensive operation since with relational algebra as the DBMS interface this involves computing a set difference. In fact, with relational algebra, the "real" work in LFP evaluation, viz., evaluating the right hand side of the recursive equations (or their differential), takes up only about 30% of the while loop execution time. The rest of the time is spent doing table copies, termination checking, and clearing temporary tables.

9. The above inefficiencies cannot be overcome using parallelism alone. While a parallel relational database machine can certainly speed up table copying and termination checking, it does not significantly reduce the percentage contribution of these operations to the while loop execution time.

10. To achieve high performance in D/KB query execution, it is very important that the relational algebra interface be augmented with a generalized LFP operator. This operator should accept a set of recursive equations of the form,

$r_i = f_i(r_1, ..., r_n)$, $i = 1, ..., n$, as input and compute their least fixed point, thereby solving each r_i. By including such an operator in the DBMS interface, many of the inefficiencies that arise with relational algebra can be alleviated. We mention several optimization possibilities that open up if the DBMS interface included an LFP operator, none of which are possible if the interface was just relational algebra:

a. Table copying can be avoided by manipulating buffer pointers.

b. The full set difference operation during termination checking can be avoided. This is because as soon as a tuple is found that was not computed in the previous iteration, the termination check can stop.

c. A dynamically adaptable indexing strategy can be designed to speed up the evaluation of the right hand side of the recursive equations or their differential. This strategy would dynamically create and drop temporary indexes on the base and intermediate derived relations depending on their relative sizes.

d. The join strategy can be dynamically changed between iterations if necessary, depending on the sized of the base and intermediate derived relations and the join selectivities from the previous iterations.

11. The performance of LFP evaluation can be significantly improved by parallel and pipelined processing. We list several strategies below:

a. During each iteration, the right hand side of each recursive equation may be evaluated in parallel.

b. Pipelining and data flow techniques may be used to evaluate the relational algebra tree corresponding to the right hand side of these equations.

c. Parallel join algorithms may be employed during this evaluation.

12. In addition to a general LFP operator, the DBMS interface should include commonly occurring special LFP operators, such as transitive closure. This is because it may be possible to optimize the execution of such special operators better than that of a general LFP operator. In general, it will be difficult for the query optimizer to recognize that a given set of LFP equations corresponds to one or another specialized LFP operator. Therefore, the Knowledge Manager interface should include ways of denoting such operators. Then the Knowledge Manager can generate code containing them and the DBMS can execute this code efficiently.

The conclusions above justify our D/KBMS architecture specification methodology. Briefly, using this methodology, we would design a high performance D/KBMS by first designing a parallel relational database machine that employs the parallel and pipelined join algorithms we developed under the VLPDF contract. Next, we would design parallel algorithms for LFP evaluation using the above join strategies, data flow and pipelining techniques, and semi-naive evaluation. Finally, we would design a Knowledge Manager that compiles Horn clause queries to relational algebra augmented with a general LFP operator and that uses the generalized magic sets strategy for restricting the search space to the relevant base relation tuples. The conclusions above suggest that

such a D/KBMS would indeed perform well.

CHAPTER 12

Conclusions and Future Directions

This chapter summarizes the conclusions from the VLPDF program and indicates several directions for future work. It is organized as follows. Sections 12.1 through 12.3 present the principal conclusions from the three phases of the VLPDF program. Section 12.4 presents our thoughts on future directions.

12.1. Phase I Conclusions

This section presents the principal conclusions from the six investigation studies conducted during Phase I.

Parallel architectures for D/KBMS application interface

- Shared memory greatly facilitates implementing stream-*AND* parallelism and the key to high performance stream-*AND* parallelism is an efficient shared memory abstraction on a loosely coupled architecture. The $(AMP)^2$ abstract machine described in chapter 3 illustrates how such an abstraction can be achieved. It does so using a number of optimizations that address critical problems in the design of efficient parallel architectures, viz., communication and memory latencies, and synchronization overheads.

- The conclusion from our work on investigating the feasibility of executing PARLOG programs on the Connection Machine architecture A coarse grained, loosely coupled architecture is better suited than the Connection Machine for executing PARLOG programs, since the form of parallelism best supported by the CM is directly opposite to that found in PARLOG.

D/KB query processing concepts

- Recursive query processing is a key concept differentiating D/KB query processing from traditional database query processing.

- The two basic strategies for Horn clause query evaluation are: top-down evaluation and bottom-up evaluation. Top-down strategies are more efficient but more complex and harder to implement. Bottom-up strategies are simpler and easy to implement but do a lot of useless work. Bottom-up evaluation of nonrecursive predicates can be accomplished via a straightforward compilation to relational algebra, while that of recursive predicates involves evaluating the LFP of a set of recursive equations.

- The two basic strategies for bottom-up LFP computation are: naive evaluation and semi-naive evaluation. Naive evaluation is more inefficient as it recomputes tuples

computed during previous iterations. Semi-naive evaluation avoid much redundant work by computing the differential of the right hand side of the recursive equations.

- Sideways information passing to restrict the search space to the relevant base relation tuples and rewriting the rules in the D/KB to an equivalent form whose LFP computation is more efficient are the basic ideas behind D/KB query optimization strategies.

Transitive closure algorithms

- Transitive closure represents an important class of D/KB queries.

- Warren's algorithm works better than the logarithmic iterative algorithm and an improved version of this algorithm in two cases: (1) the relative size of relation is not much larger than the size of available memory, and (2) the path lengths in the transitive closure graph vary greatly. In the second case, the iterative algorithms have to join two whole relations (often very large) iteratively to find a small number of tuples and the total cost increases dramatically. Thus, our recomendation is to implement Warren's algorithm for transitive closure in database systems and let the query optimizer select it adaptively.

- The HYBRIDTC algorithm that we developed and reported in chapter 5 can provide significant performance benefits, particularly in large D/KB environments, since it is very amenable to parallel processing.

Join algorithms

- It is important to choose appropriate algorithms for a particular join operation with a given system configuration. Furthermore, with a given system and relations to be joined, the query optimizer has to carefully determine the number of cluster, the number of disks and the number of processors which will be used in the join. Generally speaking, the hash-based algorithms outperform the sort-merge algorithms if the output tuples are not required in the sorted order. However, in the case that the source relations are already sorted, or the applications require the output tuples are sorted on the join attributes, the sort-merge algorithms may be advantageous.

- In multiprocessor-multidisk systems, high parallelism can be achieved by dividing the total processing task among processors and disks and executing the subtasks concurrently. However, in some algorithms, such as the sort-merge algorithms evaluated in this study, the parallel processing becomes difficult for some steps (final merge, for example). The increase of the number of processes cannot speed up the processing. On the other hand, the hash-based algorithms are naturally parallelizable. Both the partitioning and joining phase can be concurrently executed by all participating processors. This is the main reason that explains why the hash-based algorithms outperform the sort-merge algorithms with regard to the elapsed time.

- Among three major system resources. CPU, disk and communication network. CPU seems not the bottleneck of the processing pipeline in general (only in some steps of the sort-merge joins as mentioned above). For hash-based algorithms a small number of processors at each cluster is enough to provide the necessary processing power. On the other hand, disk I/O can be the bottleneck of the pipeline. although we intentionally used large page size (32K) and very high disk-memory tranfer rate in our study. One possible approach is to increase the number of disks in each cluster. This multi-disk system can efficiently remove the bottleneck caused by slow disk I/O. However. the number of disks that can be attached to one cluster must be limited by the complexity of control.

- One key point in the design of a parallel processing algorithm is to achieve maximum overlap among operations requiring different resources in order to increase the parallelism and reduce the effect of some resource which is the bottleneck of the pipeline. For example. in the hash-based algorithms. the remotely processed tuples can be transferred either during partitioning or right before their use in the joining phase. The total communication cost is the same in these two schemes. while their overlapping with disk I/O is different. In the first scheme. all communication occurs while the relations are partitioned. The second scheme distributes the communications cost: each relatively small amount of data transfer overlaps with disk I/O and CPU processing iin joining phases. Which scheme is better will depend on the relative speed of disk I/O and data transfer over the network. This example reminds us that parallelism between different type of resources can be further increased by tuning the processing steps carefully for each algorithm.

Fault tolerance

- The principal factors affecting D/KBMS availability are: system architecture. fault tolerance techniques. database size. component reliability and capacity. and data storage and access method.

- Better availability is obtained when the system has many small clusters. In most cases. $ND \leq 8$ when availability peaks.

- The effect of fault tolerance techniques on system availability is very significant. The reliability of a disk ($MTTF_{disk}$) is usually a bottleneck, and performance gain due to disk mirroring is very substantial. Processor redundancy significantly helps if disks are mirrored (or other disk redundancy methods are used). However. $NP = 2$ is sufficient for fault tolerance. A higher NP does not improve availability significantly. Fault tolerance techniques for other components are useful in conjunction with fault tolerance techniques for disks and processors.

- The size of the database has a significant effect on system availability. If the database is larger, more data will be lost when a hard failure occurs: so recovery takes longer. This degrades availability. Also. as the database size increases, the system will require more components of given capacity for storage and efficient processing. This can have a very significant effect on system availability.

- Higher $MTTF_{disk}$ improves availability significantly. System availability will also improve significantly if disks with higher capacities are used (provided $MTTF_{disk}$ of a high capacity disk is not much lower than that for a low capacity disk). A system with a higher capacity interconnect has a significantly better fault tolerance when the system consists of a few large clusters.

12.2. Phase II Conclusions

A high performance, highly available D/KBMS for very large D/KB environments can be specified using the following methodology. First design a high performance parallel relational database machine that employs the parallel and pipelined join algorithms we developed under the VLPDF contract. Next, implement the hardware and software fault tolerance mechanisms described in chapter 7. Then, design parallel algorithms for LFP evaluation using the above join strategies, data flow and pipelining techniques, and semi-naive evaluation. Finally, design a Knowledge Manager that compiles Horn clause queries to relational algebra augmented with a general LFP operator and that uses the generalized magic sets strategy for restricting the search space to the relevant base relation tuples.

12.3. Phase III Conclusions

This section presents the conclusions from the test and experimentation work in Phase III.

- In order that the D/KBMS be scalable to handle large rule sets, it is important that the rule storage structures be such that the time to extract the relevant rules is independent of the total number of rules in the Stored D/KB. Otherwise, the D/KB query compilation times will grow with the size of the rule base. Our experimentation has shown that storing the transitive closure of the PCG of the rules and placing an index on the columns of this storage structure achieves the effect of making the relevant rules extraction time independent of the rule base size.

- There are two important tradeoffs that relate to rule storage structures. The first is a time-vs-space tradeoff. Compiled form rule storage structures like the transitive closure of the PCG use more space but permit faster query compilation than non-compiled storage structures. The other tradeoff is between query compilation time and update time. Compiled form storage structures take longer to update, sometimes even an order of magnitude longer as some of our experiments indicated, than non-compiled storage structures. The choice of rule storage structure must be dictated by the relative cost of storage versus compilation and by the application characteristics — whether it is query intensive or update intensive.

- The PCG itself (as opposed to its transitive closure) is not a good choice for query intensive applications. This is because during query compilation, the transitive closure of the PCG will have to be computed to extract the relevant rules and this can get very time consuming for rules with large PCGs.

- The time to extract the relevant rules is *very* sensitive to the number of rules extracted. A key to avoiding excessive compilation times is to structure the rules

in such a way that the number of relevant rules for a query is small. Object-oriented database techniques can prove to very useful here.

- Precompilation of D/KB queries can prove to be very useful. This is especially true for frequently occurring queries with large R_{jr} values. The price of precompilation is that. for precompiled queries. information about referenced relations and rules must be recorded. During updates. this information is checked to see whether the update invalidates any compiled query. However, for applications involving few updates and frequently occurring queries with large R_{jr} values. this price is well worth paying.

- Two of the main parameters affecting D/KB query execution time are the ratio of relevant facts to total number of facts (D_{jr2}/D_{jr2}) and the amount of redundant work done in the while loop of LFP evaluation. To reduce the amount of redundant work and to restrict the LFP evaluation to the relevant database tuples. the D/KBMS architecture can use semi-naive LFP evaluation and the generalized magic sets optimization strategy.

- There is a tradeoff in using optimization: while optimization restricts LFP evaluation to the relevant tuples of the database. work must be done to first determine these tuples. Optimization pays best when the selectivity of the query is low. i.e.. for queries that retrieve only a small fraction of the database. The benefit of optimization is particularly telling for queries with very low selectivity and very large base relations.

- Relational algebra alone is not a good choice for the DBMS interface. since the LFP evaluation in this case has to be done via an application program. which introduces several inefficiencies.

- The above inefficiencies cannot be overcome using parallelism alone. While a parallel relational database machine can certainly speed up table copying and termination checking. it does not significantly reduce the percentage contribution of these operations to the while loop execution time.

- To achieve high performance in D/KB query execution. it is very important that the relational algebra interface be augmented with a generalized LFP operator.

- In addition to a general LFP operator. the DBMS interface should include commonly occurring special LFP operators. such as transitive closure.

12.4. Future Directions

This section presents several directions for future work based on the lessons learned from the VLPDF program.

- Since the system configuration. that is. the number of clusters. the number of processors. the number of disks. and the size of memory used in a join operation affects the performance along with the relation size and selectivities. query optimization in this multiprocessor environment could be more complicated. and also more important. It would be useful to more thoroughly investigate the relative behavior of different algorithms with regard to the parameters and derive some

heuristics which can be used in the query processing process.

- In our work on transitive closure algorithms, we have not assumed any auxiliary storage structures such as clustered or non-clustered indices and join indices. All operations are applied to the original data. Join indices have been shown to improve the performance of join operations. They also improve the performance of iterative transitive closure algorithms. Further investigation of the relative performance improvement of Warren's algorithm resulting from the use of auxiliary data structure is a worthwhile task.

- Depth-first and breadth-first algorithms have been explored extensively to solve the general search and tree traversal problems. Since transitive closure computation is basically a graph search problem, both depth-first and breadth first algorithms can be employed to compute the transitive closure. Warren's algorithm can be viewed as a depth-first algorithm and iterative algorithms can be viewed as breadth-first algorithms. This analogy can be useful for further research into the application of combined breadth-first and depth-first transitive closure computation techniques as has been suggested in solving other graph search problems. One possible technique is to apply an iterative algorithm a few number of iterations first to find most of the tuples in the transitive closure and then switch to the Warren's algorithm to find the few tuples which can be derived only through longer search paths.

- In general, complete transitive closure is seldom required by applications; a subset of the transitive closure is adequate for answering many queries. The algorithm to handle the restricted transitive closure queries is dependent on the restriction criteria. However, general mechanisms for restricting the output of each iteration of transitive closure operation and terminating the transitive closure computation after a specified number of iterations are possible. These mechanisms might also be useful in executing general least fixpoint queries.

- Transitive closure is a data-intensive operation. It is possible to partition the task of this very large database processing on to multiple processors and improve the performance of transitive closure computation significantly. For iterative algorithms in multiprocessor environment, the join and union operations in each iteration can be assigned to a separate processor improving the performance through concurrent and pipeline processing. For executing Warren's algorithm using multiple processors, the search of subgraphs starting from different nodes in the graph can be assigned to different processor(s). Another potential area for future work is to design, analyze and evaluate multiprocessor based iterative algorithms and Warren's algorithm.

- The HYBRIDTC algorithm described in chapter 5 has excellent potential for parallel transitive closure evaluation, but more work needs to be done. We give some suggestions below. Each processor or node can work on one or more pairs of buckets. The tuples generated at one processor are either processed locally or sent to other processors. The only synchronization needed is the final termination of the whole computation.

- Further optimization of this algorithm is worthwhile. One possibility is as follows: the new tuples generated are not only hashed on the second attribute and inserted into the corresponding buckets, but also hashed on the first attribute and inserted into the second relation in the join (RO_i^k). Thus, more tuples can be generated in each iteration, and performance improvement can be expected. However, it is somewhat difficult to implement in real system since the size of RO_i^k will change during processing. Some sophisticated memory management strategy and bucket overflow techniques have to be developed.

- Query processing and optimization techniques and the system fault tolerance are very intimately related. Query processing techniques and data storage schemes depend on the type and frequency of queries asked (i.e., the application). As we saw in our study, the data storage scheme affects fault tolerance significantly. Thus, in designing a real system, the designers of query processing software and fault tolerance have to understand the application and devise a storage scheme that is acceptable from the query processing as well as fault tolerance viewpoint. Required levels of response time, fault tolerance and reliability are application dependent.

- Define and evaluate additional fault tolerance and performance parameters. The availability measure evaluated in this report is the strong availability defined in section 7.4.2. Evaluating weak availability may give further insight into system behavior. We also think that the availability measure gives only a part of the story. Other parts of system behavior are captured by response time measures. To evaluate the system more thoroughly, additional parameters that combine no-failure response time and availability could be evaluated. Examples of such measures are average response time with failures and average system throughput [Shet87]. Evaluating these parameters may be difficult but required in light of the previous issue.

- Soft failures are tolerated using hardware methods such as processor pairing, and software methods such as transaction management and software reinitialization. Although conceptually these techniques are not difficult to understand, their implementation may be significantly difficult. We feel these techniques and their effect on overall system performance should be studied in more detail.

- We studied a data storage scheme that keeps two copies of data. If more copies of data are kept, more failures can be tolerated in general. This results in higher no-failure response time for updates but lower no-failure response time for queries. However, more data will be lost when a failure occurs. This will increase the recovery time and hence reduce the availability. A more detailed quantitative study needs to be performed to find out the optimal degree of replication.

- Fault tolerance is achieved by redundancy in hardware and software. Redundancy means additional resources and overhead. In the future, cost associated with these additional resources should be quantified with respect to the benefits of better fault tolerance. The types of fault tolerance techniques and the amount of hardware and data redundancy required should be guided by application needs.

data redundancy required should be guided by application needs.

- We found that the time taken to extract the relevant rules can be made independent of the total number of rules, but it is still remains very sensitive to the number of rules extracted. Object-oriented database techniques can prove to be very useful in reducing the time taken to extract the relevant rules during D/KB query compilation since they provide efficient structuring of the D/KB. For example, a small set of rules can be encapsulated within an object and these rules can be retrieved whenever the object receives a message representing a query against them. Encapsulating rules within an object is a way of structuring the rules so that only the relevant portions of the rule base are processed during compilation. Of course, much work needs to be done to integrate the concepts of object-oriented database systems — inheritance, message processing, persistent objects. etc. — with those of D/KB query and update processing.

- We found that D/KB query compilation can constitute a significant portion of D/KB query processing time. Techniques for handling precompiled D/KB queries need to be integrated into the D/KBMS architecture specification methodology.

- Efficient implementation techniques for generalized LFP operators should be investigated. We mention several optimization possibilities below: an LFP operator. none of which are possible if the interface was just relational algebra:

  a. Table copying can be avoided by manipulating buffer pointers.

  b. The full set difference operation during termination checking can be avoided. This is because as soon as a tuple is found that was not computed in the previous iteration. the termination check can stop.

  c. A dynamically adaptable indexing strategy can be designed to speed up the evaluation of the right hand side of the recursive equations or their differential. This strategy would dynamically create and drop temporary indexes on the base and intermediate derived relations depending on their relative sizes.

  d. The join strategy can be dynamically changed between iterations if necessary. depending on the sized of the base and intermediate derived relations and the join selectivities from the previous iterations.

- Parallel algorithms for general LFP evaluation can significantly improve D/KB query execution performance. We list several strategies below:

  a. During each iteration. the right hand side of each recursive equation may be evaluated in parallel.

  b. Pipelining and data flow techniques may be used to evaluate the relational algebra tree corresponding to the right hand side of these equations.

  c. Parallel join algorithms may be employed during this evaluation.

References

Agra87. R. Agrawal, "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries," Third International Conference on Data Engineering, pp. 580-590, IEEE, Los Angeles, February 3-5, 1987.

Aho79. A. V. Aho and J. D. Ullman, "Universality of Data Retrieval Languages," Proc. Sixth ACM Symp. on Principles of Programming Languages, pp. 110-117, Jan. 1979.

Arvi82. Arvind and K. P. Gostelow, "The U-Interpreter," *Computer*, vol. 15, no. 2, pp. 42-49, Feb 1982.

Aviz84. A. Avizienis and P. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, vol. 17, no. 8, August 1984.

Banc85. F. Bancilhon, "Naive Evaluation of Recursively Defined Relations," in *On Knowledge Base Management Systems — Integrating Database and AI Systems*, ed. Brodie and Mylopoulos, Springer-Verlag, 1985.

Banc86. F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," Proc. SIGMOD, Invited Paper, 1986.

Banc86.......F. Bancilhon *et al.*, "Magic Sets and Other Strange Ways to Implement Logic Programs," Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.

Barr81. A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence, Vol. I*, William Kaufmann, Inc., Los Altos, CA, 1981.

Beer86. C. Beeri and R. Ramakrishnan, "On the Power of Magic," MCC Technical Report, DB-061-86, also submitted to Principles of Distributed Systems, 1987, November, 1986.

Bern85. P. A. Bernstein, "Sequoia: A Fault-Tolerant Tightly-Coupled Computer for Transaction Processing," Technical Report TR-85-03, Wang Institute of Graduate Studies, May 2, 1985.

Bitt83. D. Bitton *et al.*, "Parallel Algorithms for the Execution of Relational Database Operations," *ACM TODS*, vol. 8, no. 3, pp. 324-353, Sept. 1983.

Borr81. A. Borr, "Transaction Monitoring in Encompass," 7th VLDB, September 1981. Also Tandem Computers TR 81.2

Borr84. A. Borr, "Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-processor Approach," 9th VLDB, September 1984. Also Tandem Computers TR 84.2

Brat84. K. Bratsbergsengen, "Hashing Methods and Relational Algebra Operations," Proc. 10th VLDB, pp. 323-333, Singapore, August 1984.

Brod86. Michael L. Brodie *et al.*, "Knowledge Base Management Systems: Discussions from the Working Group," in *Expert Database Systems, Proceedings From the First International Workshop*, ed. Larry Kerschberg, pp. 19-33, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1986.

Bune82.   P. Buneman, R. E. Frankel, and R. Nikhil, "An Implementation Technique for Database Query Languages," *ACM TODS*, vol. 7, no. 2, pp. 164-186, June 1982.

Butl85.   R. Butler *et al.*, *Parallel Logic Programming for Numeric Applications*, Argonne National Laboratory, 1985.

Clar86.   K. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," *ACM TOPLAS*, vol. 8, no. 1, pp. 1-49, Jan. 1986.

Clar81.   K. L. Clark and S. Gregory, "A Relational Language for Parallel Programming," Proc. 1981 ACM Conf. on Functional Programming Languages and Computer Architecture, pp. 171-178, Portsmouth, NH, Oct. 1981.

Cloc84.   W. F. Clocksin and C. S. Mellish, *Programming in Prolog, 2nd ed.*, Springer-Verlag, Berlin, 1984.

DeWi79.   D.J. DeWitt, "DIRECT — A Multiprocessor Organization for Supporting Relational Database Management Systems," *IEEE Trans. Computers*, vol. C-28, no. 6, 1979.

DeWi84.   D. J. DeWitt *et al.*, "Implementation Techniques for Main Memory Database Systems," Proc. SIGMOD '84, pp. 1-8, ACM, June 1984.

DeWi85.   D. J. DeWitt and R. Gerber, "Multiprocessor Hash-Based Join Algorithms," Proc. VLDB 85, pp. 151-164, Stockholm, Aug. 1985.

DeWi86.   D.J. DeWitt *et al.*, "GAMMA — A High Performance Dataflow Database Machine," Proc. VLDB 86, pp. 228-237, Kyoto, Japan, August, 1986.

Deck86.   J. Decker, "C3I Teradata Study," (RADC-TR-85-273), Rome Air Development Center, Griffiss Air Force Base, NY, March 1986.   (A169300)

Emde76.   M. H. van Emden and R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language," *JACM*, vol. 23, no. 4, pp. 733-742, Oct 1976.

Forg81.   C. L. Forgy, "OPS5 User's Manual," Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie Mellon University, July 1981.

Gray86.   J. Gray, "Why Do Computers Stop and What Can Be Done About It?," Proc. 5th Symposium on Reliability in Distributed Software and Database Systems, pp. 3-11, 1986.

Greg85.   S. Gregory, "Design, Application and Implementation of a Parallel Logic Programming Language, PhD Dissertation," *Imperial College of Science and Technology, University of London, Department of Computing*, September 1985.

Haed83.   T. Haeder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, vol. 16, no. 4, December 1983.

Hali84.   M.Z. Halim, "Data-driven and Demand-driven Evaluation of Logic Programs," *PhD Thesis, Department of Computer Science, University of Manchester*, 1984.

Hamm82.   P. Hammond, "APES (A Prolog Expert System Shell): A User Manual," Research Report DOC 82/9, Department of Computing, Imperial College London, London, 1982.

Hill85.   D. Hillis, *The Connection Machine*, The MIT Press, Cambridge, MA, 1985.

Iann83.   Arvind, R. A. Iannucci, "A Critique of Multiprocessing von Neumann Style," *10th. Computer Architecture Symposium*, 1983.

Ioan86.   Y.E. Ioannidis, "On the Computation of the Transitive Closure of Relational Operations," Proc. 12th International Conf. on VLDB, Kyoto, Japan, August 1986.

Ito85.   N. Ito, et.al., "The Dataflow-based Parallel Inference Machine to Support Two Basic Languages in KL1," *Technical Report, ICOT*, Tokyo, 1985.

Kast83.   P. S. Kastner, "A Fault-Tolerant Transaction Processing Environment," *Database Engineering Bulletin*, vol. 6, no. 2, June 1983.

Kim84.   W. Kim, "Highly Available Systems for Database Applications," *ACM Computing Surveys*, vol. 16, no. 1, pp. 71-98, March 1984.

Kits83.   M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Application of Hash to Data Base Machine and its Architecture," *New Generation Computing*, vol. 1, no. 1, pp. 63-74, 1983.

Knut73.   D. E. Knuth, *The Art of Computer Programming, Volume 3 (Sorting and Searching)*, Addison-Wesley, Reading, MA, 1973.

Koon86.   T. Koon and M. Tamer Ozsu, "Performance Comparison of Resilient Concurrency Control Algorithms for Distributed Databases," 2nd Intl. Conf. on Data Engineering, Los Angeles, February 1986.

Lu85.   H. Lu and M.J. Carey, "Some Experimental Results on Distributed Join Algorithms in a Local Network," Proc. VLDB 1985, Stockholm, August 1985.

Lu87.   H. Lu, K. Mikkilineni, and J. Richardson, "Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation," Proc. 3rd Internationsl Conf. on Data Engineering, Los Angeles, CA, February 3-5 1987.

McKa81.   D. P. McKay and S. C. Shapiro, "Using Active Connection Graphs for Reasoning with Recursive Rules," Proc. Seventh International Conf. on Artificial Intelligence, pp. 368-374, Aug. 1981.

Mill84.   J. A. Crammond, C. D. F. Miller, "An architecture for parallel logic languages," *Proceedings of the 2nd International Logic Programming Conference*, pp. 183-194, Uppsala, July 1984.

Rose86.   A. Rosenthal *et al.*, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," Proc. 1986 ACM-SIGMOD Conference, Washington, D.C., May 1986.

Sacc86.   D. Sacca and C. Zaniolo, "On the Implementation of a Simple Class of Logic Queries for Databases," 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.

Sacc86........D. Sacca and C. Zaniolo, "The Generalized Counting Method for Recursive Logic Queries," Proc. ICDT, 1986.

Schl83.   R. Schlichting and F. Schneider, "Fail-Stop Processors, an Approach to Designing Fault-Tolerant Computing Systems," *ACM TOCS*, vol. 1, no. 3, August 1983.

Schu79.   S.A. Schuster *et al.*, "RAP.2 — An Associative Processor for Databases and its Applications," *IEEE Trans. Computers*, vol. C-28, no. 6, 1979.

Shap83.   E. H. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," Technical Report TR-003, ICOT, Feb 1983.

Shet85.   A. Sheth, A. Singhal, and M. Liu, "An Analysis of the Effect of Network Parameters on the Performance of Distributed Database Systems," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1174-1183, October 1985.

Shet87.   A. Sheth, A. Singhal, and M. Liu, "Performance Analysis of Resiliency Mechanisms in Distributed Database Systems," Proc. 3rd Intl. Conf. on Data Engineering, February 1987.

Shib84.   S. Shibayama *et al.*, "A Relational Database Machine with Large Semiconductor Disk and Hardware Relational Algebra Processor," *New Generation Computing*, vol. 2, no. 2, pp. 131-155, ? 1984.

Shor76.   E. H. Shortliffe, *Computer-based Medical Consultations: MYCIN*, North Holland, New York, 1976.

Siew82.   D. Siewiorek and R. Swarz, "The Theory and Practice of Reliable System Design," in *Digital Press*, Bedford, MA, 1982.

Siew84.   D. Siewiorek, "Architecture of Fault-Tolerant Computers," *IEEE Computer*, vol. 17, no. 8, August 1984.

Siew78.   D. P. Siewiorek *et al.*, "A Case Study of C.mmp, Cm*, and C.vmp: Part 1-Experiences with Fault Tolerance in Multiprocessor Systems," *Proc. IEEE*, vol. 66, no. 10, pp. 1178-1199, October 1978.

Smit86.   M. Smith, personal communication, October 1986.

Stef82.   M. Stefik *et al.*, *The Organization of Expert Systems: A Prescriptive Tutorial*, Xerox Palo Alto Research Center, Palo Alto, CA, Jan. 1982.

Su79.   S.Y.W. Su *et al.*, "The Architectural Features and Implementation Techniques of the Multicell CASSM," *IEEE Trans. Computers*, vol. C-28, no. 6, 1979.

Subr85.   P. A. Subrahmanyam, "The "Software Engineering" of Expert Systems: Is Prolog Appropriate?," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 11, pp. 1391-1400, Nov 1985.

Tars55.   A. Tarski, "A Lattice-Theoretical Fixpoint Theorem and its Applications," *Pacific J. Mathematics*, vol. 5, no. 2, pp. 285-309, June, 1955.

Tera83.   Teradata CorporationTeradata, *DBC/1012 Data Base Computer Concepts and Facilities*, Inglewood, CA, Apr. 1983.

Thom80.   Arvind, R. E. Thomas, "I-structures: An Efficient Data Type for Functional Languages," *MIT/LCS/TM-178*, September 1980.

Ullm85.   J. D. Ullman, "Implementation of Logical Query Languages for Databases," *ACM TODS*, vol. 10, no. 3, pp. 289-321, Sept 1985.

Vald84.   P. Valduriez and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine," *ACM TODS*, vol. 9, no. 1, pp. 133-161, March 1984.

Vald86.   P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices," Proc. First Intl. Conf. on Expert Database Systems, Apr. 1986.

Warr75.   H. S. Warren, "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *CACM*, vol. 18, no. 4, pp. 218-220, Apr. 1975.

Wars62.   S. Warshall, "A Theorem on Boolean Matrices," *JACM*, vol. 9, no. 1, pp. 11-12, Jan. 1962.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.